DISSERTATIONES
MATHEMATICAE
UNIVERSITATIS
TARTUENSIS
114

ALISA PANKOVA

Efficient Multiparty Computation Secure against Covert and Active Adversaries

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
114

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
114

ALISA PANKOVA

Efficient Multiparty Computation Secure against Covert and Active Adversaries

UNIVERSITY OF TARTU
Press

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia

Dissertation is accepted for the commencement of the degree of Doctor of Philosophy (PhD) on May 4, 2017 by the Council of the Institute of Computer Science, University of Tartu.

Supervisor:

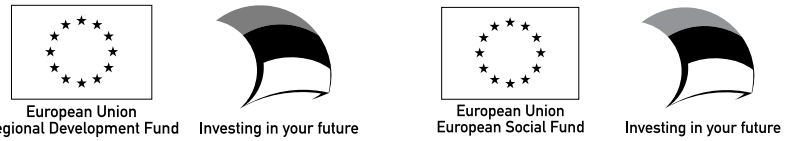
Ph.D	Peeter Laud Cybernetica AS Tartu, Estonia
Dr.Tech	Sven Laur University of Tartu Tartu, Estonia

Opponents:

PhD	Joël Alwen Institute of Science and Technology Austria Klosterneuburg, Austria
PhD	Jun Furukawa NEC Corporation America Herzliya, Israel

The public defense will take place on June 22, 2017 at 10:15 in Liivi 2-403.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu.



ISSN 1024-4212
ISBN 978-9949-77-443-2 (print)
ISBN 978-9949-77-444-9 (PDF)

University of Tartu Press
www.tyk.ee

Contents

ABSTRACT

Secure multiparty computation is one of the most important employments of modern cryptography, bringing together elegant mathematical solutions to build up useful practical applications. It allows several distinct data owners to perform arbitrary collaborative computation on their private data without leaking any information to each other.

The security of multiparty computation often relies on assumptions about the behaviour of parties. A passively secure protocol is secure as long as all parties follow its rules. A covertly secure protocol works under assumption that no party will cheat if it will be detected by the other parties. An actively secure protocol is able to tolerate any behaviour of corrupted parties.

This thesis presents a generic method for turning passively secure multiparty protocols to covertly or actively secure ones, assuming that the majority of parties is honest. The method is optimized for three party computation over rings of residue classes $\mathbb{Z}_{2^n}$, which has proven to be quite an efficient model, making large real-world applications feasible.

In this thesis, we also study a new adversarial goal in multiparty protocols. The goal is to manipulate the view of some honest party in such a way, that this honest party learns the private data of some other honest party. The adversary itself might not learn this data at all. Such attacks are significant because they create a liability to the first honest party to clean its systems from the second honest party's data, which may be a highly non-trivial task in practice.

Finally, this thesis addresses the problem of excessive computation in secure multiparty applications. In some cases, the parties need to make a decision, in which direction their computation should proceed further. If this decision depends on private data, then the parties are not allowed to know which computational branch has been chosen, so in general the parties need to execute all of them. If the number of branches is large, the computational overhead may be enormous, as most of the intermediate results will not be needed for the final answer. This thesis proposes an optimization that reduces this overhead.

CHAPTER 1

INTRODUCTION

1.1 Secure Multiparty Computation

Secure multiparty computation is a methodology that allows several parties to process private data collaboratively without revealing the data to any party. The data are protected by encryption or some other similar method such as secret sharing. Some possible scenarios of secure multiparty computation are the following:

- A client uses external resources (e.g. a cloud) to process data collected from multiple data owners. The data should be protected from both the resource controller and the client.
- Multiple clients collaborate to process data provided by themselves, protecting their own data from each other.
- Multiple clients collaborate to process data provided by multiple data owners. The data should be protected from each client.

First research results on secure multiparty computation have been published in 1980-s [104]. Numerous theoretical solutions and implementation prototypes have been proposed since that time [47, 8, 75]. It took time for secure multiparty computation to become practical, and the first really big application of secure multiparty computation on real data was Danish sugar beet auction [19] that took place in 2008, where the task of sugar beet providers and their customer was to agree on the selling price.

Much work on practical use of secure multiparty computation on real data has been accomplished since that time. A secure system for jointly collecting and analyzing private financial data was implemented and deployed in 2011 [18]. A tax fraud detection system that works with private data has been implemented and evaluated in 2014 [11]. The relationship between working during university

studies and graduating on time has been studied in 2015 [12], and it was required to combine private data of the Ministry of Education and the Tax and Customs Board. Secure survey systems have been used in large scale in 2015-2016, when Boston Women's Workforce Council initiated a study of gender and ethnicity wage gaps among employers within the Greater Boston Area [61].

Secure multiparty computation is increasingly being used in the real world scenarios, and hence it is important to contribute to this area, so that it could be used in even a better way.

1.2 Assumptions of Secure Multiparty Computation

Secure multiparty computation often makes some assumptions about the behaviour of computing parties, i.e. the parties that actually perform the computation, who are not necessarily the data owners. One possible assumption is the guaranteed number of honest parties. If the inputs are protected by secret sharing them among the parties, then existence of at least one honest party should be assumed in any case. Otherwise, if all the computing parties colluded, they would recover the secret even if no computation took place at all. Making a bit stronger assumptions, such as an honest majority of parties, may result in much more efficient protocols.

Assumptions are also being made about the behaviour of dishonest parties. Passively corrupted parties are curious, and may collude to recover the secret, but they do not violate the protocol rules. Actively corrupted parties may also collude, and in addition they may deviate from the protocol rules. Protocols secure against actively corrupted parties (*actively secure* protocols) are relatively efficient nowadays, but they still underperform severely when compared to the protocols that are secure only against passively corrupted parties (*passively secure* protocols). If data are big, execution of an actively secure protocol on these data may be infeasible.

When a passively secure protocol is used, data protection is guaranteed only if none of the computing parties will actually try to break the rules deliberately. This is often a reasonable assumption, depending on who the computing parties are. If a party is some honorable institution, then it would take care of its reputation and never try to cheat if there was any possibility that the cheating could be detected. A party that will not cheat if it will be detected by the other parties is called covertly corrupted. In general, passively secure protocols do not attempt to detect such cases. Even if all the other parties notice that something goes wrong, it is quite unlikely that the cheater will be identified.

To get a protocol that would be secure against covertly corrupted parties, it is sufficient to extend a passively secure protocol with a verification phase such that, if any party has cheated during the execution, it would be noticed by all the other

parties. It would be even better if the computation could be verified publicly, so that the misbehaviour of the cheater could be proven to an external judge. Looking for new efficient methods to achieve accountable security is a promising research area. In this thesis, we investigate this area and propose a verification method that works under honest majority assumption and is quite efficient for a small number of parties.

One problem of the existing security models of multiparty protocols is that they do not cover well the ability of the attacker to leak the data of one honest party to another honest party. Such attacks are possible, even if the attacker does not see these data himself. This is not a problem if the protocol does not make any assumptions on the number of honest parties, since any party may potentially be corrupted, so the protocol should protect data from all of them. However, if there is an assumption that the number of corrupted parties is bounded, then the models take into account only the data leaked to these corrupted parties. Little research has been done in this direction so far, and we investigate the problem in this thesis.

1.3 Claims of This Thesis

The aim of this thesis is to weaken some set-up assumptions of secure multiparty computation while keeping efficiency on a level that allows to apply it to real data. In particular, this thesis proposes a method for making passively secure protocols efficiently verifiable, and a method for preventing the honest parties from learning too much. On the other hand, it does not attempt to get rid of the honest majority assumption. This thesis states and proves the following claims:

Claim 1: One can transform passively secure multiparty protocols over rings of residue classes $\mathbb{Z}_n$ to protocols that are actively secure under the honest majority assumption, introducing a small overhead. The resulting protocols are especially efficient for three parties and for rings $\mathbb{Z}_{2^m}$.

Claim 2: There exists a security model that allows detecting the leakage of one honest party's data to another honest party. The simplicity of proofs in this model is similar to the existing widely used security models that fail to detect such leakage, and achieving security in this model is feasible in practice.

Claim 3: The overhead of secure computation applications, caused by the necessity of computing all branches of choices that depend on private data, can be reduced by an optimization that does not affect security of the initial application.

1.4 Outline and Author's Contributions

We explain the notation and the basics needed to understand our work in Chapter 2. We discuss the related work in Chapter 3. The subsequent chapters comprise the actual contribution of this thesis.

Contribution 1: In Chapter 4, we construct a protocol set for verifying secure multiparty computation. The verification property is very strong, and a misbehaving party will remain undetected at most with negligible probability. This allows to turn passively secure protocols to covertly secure (if the verification is applied once), or actively secure (if the verification is applied after each protocol round). Compared to other similar verification methods from related work, the specific feature of our mechanism is its straightforward compatibility with computation over rings, and its efficiency in the case of three parties. We apply our verification mechanism to the protocol set [17] employed in the Sharemind platform [16], demonstrating for the first time a method to achieve active security for Sharemind.

Chapter 4 proves Claim 1 of this thesis. This chapter is based on a previous publication of the author [62], and an unpublished result [63].

- Laud, P., Pankova, A.: Verifiable Computation in Multiparty Protocols with Honest Majority. In: Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, China. Proceedings. LNCS, vol. 8782, pp. 146–161. Springer (2014).
- Laud, P., Pankova, A.: Preprocessing-based Verification of Multiparty Protocols with Honest Majority. Cryptology ePrint Archive, Report 2015/674.

Contribution 2: In Chapter 5, we define a new adversarial model that allows to capture protocol vulnerabilities resulting in leaking information to *honest* parties. We prove that our model is as strong as the standard *universal composability* [21] model, and that it discovers attacks that are not captured by alternative models from related work. As an example, we study our verification mechanism in this new model, find the vulnerabilities in our protocols, and show how to fix them. We estimate the overheads caused by this improvement, showing that it is feasible to achieve security in the new model without complicated constructions.

Chapter 5 proves Claim 2 of this thesis. This chapter is based on a previous publication of the author [65].

- Laud, P., Pankova, A.: Securing multiparty protocols against the exposure of data to honest parties. In: Data Privacy Management and Security Assurance - 11th International Workshop, DPM 2016 and 5th International Workshop, QASA 2016, Heraklion, Crete, Greece, Proceedings. LNCS, vol. 9963, pp. 165–180. Springer (2016)

Contribution 3: In Chapter 6, we propose an optimization for secure computation, applicable to both execution and verification of computation. The goal of the optimization is to reduce the number of computations whose results are not needed for the final answer, but which unfortunately have to be introduced in some cases to ensure data privacy. We estimate performance and usefulness of our optimization algorithms on the example of the Sharemind platform [16].

Chapter 6 proves Claim 3 of this thesis. This chapter is based on a previous publication of the author [64].

- Laud, P., Pankova, A.: Optimizing secure computation programs with private conditionals. In: Information and Communications Security - 18th International Conference, ICICS 2016, Singapore, Proceedings. LNCS, vol. 9977, pp. 418–430. Springer (2016),

CHAPTER 2

PRELIMINARIES

2.1 Multiparty Computation

In multiparty computation, there are n distinct parties with mutually disjoint inner states that compute a certain functionality f on their inputs $x_1, \ldots, x_n$, getting the outputs $y_1, \ldots, y_n = f(x_1, \ldots, x_n)$. The parties are connected with each other via channels that they use to exchange messages. The i-th party performs some local computations on x_i and the messages it receives from the other parties. As the result, it gets messages that it sends to the other parties, and the output y_i. The value of y_i in general depends not only on x_i, but on all the inputs $x_1, \ldots, x_n$, as well as on randomness of the parties. A multiparty *protocol* is a set of rules specifying which computation should be performed by which party, and which messages should be exchanged.

The parties can represent different physical machines in the network, e.g. the client and the server, or just different computational units in the same machine, e.g. interaction of a trusted computation hardware unit [77, 48, 2] with an untrusted memory. A single party may also represent a set of machines. In this thesis, we treat parties as logical entities, without going into detail of their implementation.

Circuits. The functionality f that the parties collaboratively compute, as well as the local computations of each party, are often formalized by arithmetic or boolean *circuits*. A circuit is a directed acyclic graph, whose nodes (*gates*) represent the computational operations, and whose arcs (*wires*) denote the input/output relations between the gates. Each gate computes some operation on the values coming from its incoming arcs (*input wires*), and propagates the result to all its outgoing arcs (*output wires*). Some wires may have only one endpoint. The wires that have no source are called *circuit inputs*, and the wires that have no target are called *circuit outputs*.

An *arithmetic circuit* consists of addition, multiplication by a constant, multiplication of two variables, and constant gates, all defined over a ring or a field. A *boolean circuit* consists of exclusive disjunction (XOR), conjunction (AND), negation (NOT), and constant gates, defined over the boolean values $\{0, 1\}$. It is known that both types of circuits are sufficient to represent any computation.

Rounds and Synchronicity. Computation between parties takes place in *rounds*. A round covers the parties' computation that takes place between the receipt of the inputs or messages of the previous round, and the output of the messages of the next round to the other parties. The computation of parties can be represented by a set of circuits C_{ij}^{ℓ} computing the messages that a party P_i sends to the party P_j on the ℓ-th round. The communication is *synchronous* if all the messages of the ℓ-th round are output before any message of the $\ell + 1$-st round is output.

2.2 Security of Multiparty Computation

2.2.1 Secrets

The inputs of parties in multiparty computation can be private. To deal with this, multiparty computation can be constrained to *secure multiparty computation* (SMC), where the initial inputs of the parties, and often also their outputs, are treated as secrets.

In SMC protocol sets based on *secret sharing* [47, 26, 17, 33], the involved parties are usually partitioned into input, computation, and output parties [90]. The input parties provide the inputs by secret-sharing them among the computation parties. The computation parties carry out computation on these shares to obtain the desired result. The output parties receive the shares of final outputs, which they can recombine into the final outputs of the computation. In this case, the actual secrets are the values that have been provided by the input parties, and the values that are finally given to the output parties. No computation party should be able to infer any information about these values, and this property must be ensured by the used secret sharing scheme. We discuss some of these schemes in Section 2.4.

2.2.2 Adversary

Intuitively, the adversary is an evil entity that tries to break the computation, and/or extract some secret information out of it. We need to define more precisely where this evil entity is residing in the SMC setting. Assuming that there are secure authenticated channels between the parties eliminating man-in-the-middle attacks, the only possible attackers are the parties themselves. Some parties may try to infer the other parties' inputs from the messages they get, or even try to break

the protocol by sending invalid messages on their own behalf. As it is possible that different parties will collaborate to conduct stronger attacks, the adversary is usually viewed as a single entity controlling all the parties that it *corrupts*. The allowed upper bound on the number of corrupted parties is defined explicitly for each SMC protocol.

A *static* adversary corrupts the parties in advance before the protocol starts. An *adaptive* adversary may corrupt parties at any time during protocol execution, possibly releasing them at some point, in order to corrupt some other parties while maintaining the upper bound on the total number of currently corrupted parties. In this thesis, we only consider static adversaries.

Adversaries can also be differentiated based on the behaviour of parties that they corrupt.

- **Passive (semihonest, honest-but-curious):** the corrupted party follows the protocol as an honest party, but it shares its internal state with the adversary.
- **Covert [4]:** the corrupted party may misbehave, but only as far as the probability of being caught is negligible. By being caught we mean that all honest parties of the protocol unanimously agree that this party is guilty.
- **Fail-Stop [42]:** the corrupted party follows the rules, but at some moment it may try to stop the protocol, so that the computation fails. In this thesis, we use the definition where the party may stop the protocol only if the probability of being caught is negligible. In this way, we consider fail-stop adversary as an instance of covert adversary.
- **Active (malicious):** the corrupted party does whatever it wants.

The passive and the active adversary are the two main kinds of adversaries for SMC protocols that are typically considered. The highest performance and the greatest variety is achieved in protocols secure against passive adversaries. In practice we would like to achieve stronger security guarantees. Achieving security against active adversaries may be expensive. Hence, intermediate classes between passive and active adversaries have been introduced. In this thesis, we pay special attention to the covert adversary.

2.2.3 Basics of Universal Composability

In this thesis, we study security of our protocols in the *universal composability* (UC) model [21]. This model considers systems of interactive Turing machines (ITM) connected to each other by input and output communication tapes and maintaining some internal state. Throughout this thesis, in the figures, ITMs are represented by boxes, and the communication tapes by arrows.

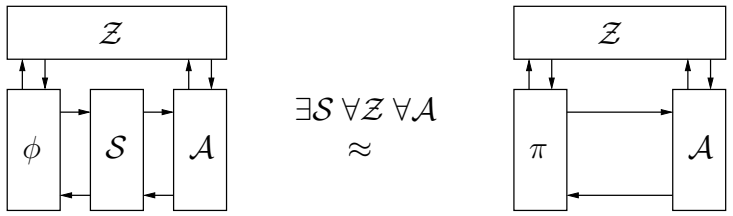


Figure 2.1: UC-emulation: the protocol π UC-emulates the protocol ϕ

Model Description

A protocol Π consists of ITMs P_i, where i is a unique identifier. The protocol represents the computing parties that mutually realize some functionality $\mathcal{F}$. The parties may be connected to each other, and may also use a trusted resource ITM R to mediate their communication or even compute something for them. A special ITM $\mathcal{A}$ represents the *adversary* that may corrupt some parties P_i and get access to their internal states. There is a special ITM $\mathcal{Z}$, the *environment*, that chooses the inputs for each P_i and receives their outputs. $\mathcal{Z}$ may represent the users sitting behind the machines P_i, as well as any other protocols running concurrently with Π, probably even some other sessions of Π. There is also communication between $\mathcal{Z}$ and $\mathcal{A}$.

In security proofs, one defines an ideal functionality $\mathcal{F}$ represented by a trusted ITM. It receives the inputs from all parties, computes some function on these inputs, and returns to each party its output. It may deliberately output to the adversary some data that is insensitive enough to be leaked. On the other hand, there is a protocol Π that shares exactly the same communication tapes with $\mathcal{Z}$ as $\mathcal{F}$ does, but that consists of untrusted machines P_i communicating with each other, and optionally some other resource R used by these machines. The settings of Π are much more realistic than $\mathcal{F}$, and the goal is to show that Π is secure enough to be used instead of $\mathcal{F}$. This can be done by proving that any attack (represented by $\mathcal{A}$) against Π can be converted to an attack (represented by some *ideal* adversary $\mathcal{A}_S$) against $\mathcal{F}$. Formally, one proves that no environment $\mathcal{Z}$ is able to distinguish whether Π with $\mathcal{A}$ or $\mathcal{F}$ with $\mathcal{A}_S$ is running, regardless of the adversary $\mathcal{A}$. The proof is done by finding a suitable adversary $\mathcal{A}_S$, and it is often defined as $\mathcal{A}_S = (\mathcal{S}\|\mathcal{A})$ for the *simulator* $\mathcal{S}$ that mediates the communication between $\mathcal{A}$ and $\mathcal{F}$, trying to convince $\mathcal{A}$ that it is communicating with Π, and trying to convince $\mathcal{F}$ that it is communicating with $\mathcal{A}_S$. The ability of $\mathcal{Z}$ to distinguish between the two protocols is captured by defining the *final output* of $\mathcal{Z}$. The distributions of the final outputs of $\mathcal{Z}$ should be indistinguishable for $\mathcal{F}$ and Π, regardless of the values that $\mathcal{Z}$ actually outputs. Let $EXEC_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the probability ensemble of outputs of the environment $\mathcal{Z}$ running the protocol Π with the adversary $\mathcal{A}$.

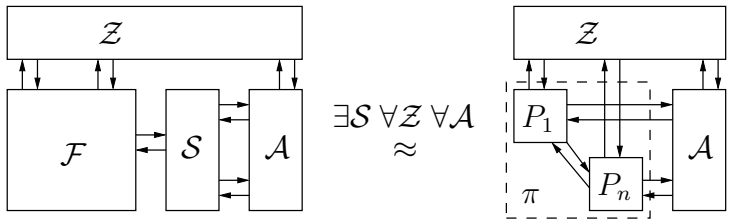


Figure 2.2: UC-realization: the protocol π UC-realizes the ideal functionality $\mathcal{F}$

Definition 2.1 (UC emulation [21])**.** Let π and ϕ be probabilistic polynomial time (PPT) protocols. We say that π UC-emulates ϕ if there exists a PPT machine S, such that for any PPT adversary $\mathcal{A}$, and for any PPT environment $\mathcal{Z}$, the probability ensembles $EXEC_{\pi,\mathcal{A},\mathcal{Z}}$ and $EXEC_{\phi,(S\|\mathcal{A}),\mathcal{Z}}$ are indistinguishable (denoted $EXEC_{\pi,\mathcal{A},\mathcal{Z}} \approx EXEC_{\phi,(S\|\mathcal{A}),\mathcal{Z}}$).

UC-emulation is depicted in Figure 2.1. If the protocol ϕ is defined in such a way that forwarding messages between $\mathcal{Z}$ and the subroutine $\mathcal{F}$ is the only thing that the parties do, we may also say that the protocol π UC-*realizes* $\mathcal{F}$. This is depicted in Figure 2.2. We give some formal interpretations of indistinguishability in Section 2.3.1, explaining why we require the machines $\mathcal{A}$, $\mathcal{S}$, $\mathcal{Z}$ to be PPT.

Since Definition 2.1 does not specify the adversary type, we will further explicitly specify whether a protocol emulates the functionality passively, covertly, or actively. If there are no additional adversary specifications, then by default an active adversary is assumed.

In order to make the proofs of this thesis simpler, we modularize them, proving the following two properties separately:

1. **Simulatability:** the simulator $\mathcal{S}$ is able to simulate to $\mathcal{A}$ all the messages sent to the corrupted parties, so that they come from the same distribution as if Π was run on the same party inputs.

2. **Correctness:** the party outputs in the ideal functionality $\mathcal{F}$ are exactly the same as in the simulated Π. This is true also for the probabilistic outputs.

If these properties hold, then $\mathcal{Z}$ cannot distinguish between $\mathcal{F}$, $\mathcal{A}_S$ and Π, $\mathcal{A}$, even if it puts together the messages it receives from $\mathcal{F}$ and $\mathcal{A}_S$ (Π and $\mathcal{A}$).

Composition Theorem

An important feature of the UC model is that the proofs are extended automatically to protocol compositions. Namely, proving that a protocol π UC-emulates ϕ is sufficient to substitute any instances of ϕ in a complex protocol with π, without losing in security, even if multiple instances of ϕ are running in parallel or sequentially.

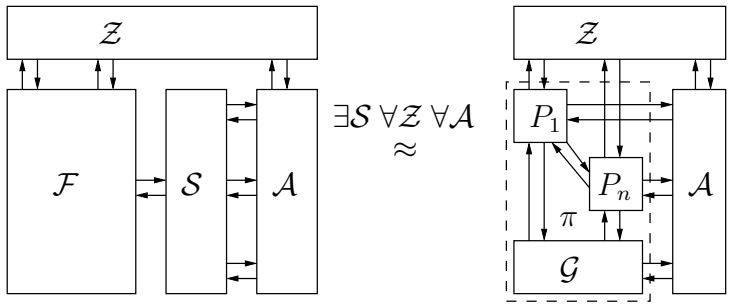


Figure 2.3: UC hybrid model: the protocol π UC-realizes $\mathcal{F}$ in $\mathcal{G}$-hybrid model

Theorem 2.1 (UC composition theorem [21])**.** *Let ρ, ϕ, π be protocols such that ρ uses ϕ as subroutine, and π UC-emulates ϕ. Then $\rho[\phi \to \pi]$ (a protocol that results from substituting ϕ with π in ρ) UC-emulates ρ.*

Theorem 2.1 allows the protocol π (whose security is being proven) to use some ideal functionality $\mathcal{G}$ as a subroutine, assuming that, in the actual implementation, $\mathcal{G}$ will be substituted with some protocol UC-realizing it. In this case, we say that the proof of π UC-realizing an ideal functionality $\mathcal{F}$ is done in so-called *hybrid model*. UC-realization in the hybrid model is depicted in Figure 2.3.

Generalized UC

The proofs in the UC model make an assumption that each protocol runs its own instance of each subroutine that it uses, and these subroutines cannot be accessed by any other protocol. This gives the simulator full control over the subroutines. For example, if a protocol uses a subroutine that models public key infrastructure, then the proofs assume that a new set of public/private keys is generated each time for each protocol execution, although in reality the same keys would be reused by different protocols.

An extension of UC, called *generalized* UC (GUC [22]) was proposed to model such a global trusted setup. In this model, the keys can be generated by a *shared functionality* that can be accessed by different protocols, or multiple sessions of the same protocol. In general, GUC allows to use any subroutine as a shared functionality. If $\mathcal{F}$ is a functionality used as a subroutine, then $\underline{\mathcal{F}}$ denotes a shared instance of the same functionality. Formally, all these outer protocols using $\underline{\mathcal{F}}$ can be treated as a part of the environment. In this way, the GUC model can be seen as a generalization of UC, where the environment is not constrained to invoking only one session of the protocol that it is attempting to distinguish.

Even if there is just one shared functionality $\underline{\mathcal{G}}$ used by the protocol, the proofs in the plain GUC model are quite complex as there can be an unbounded

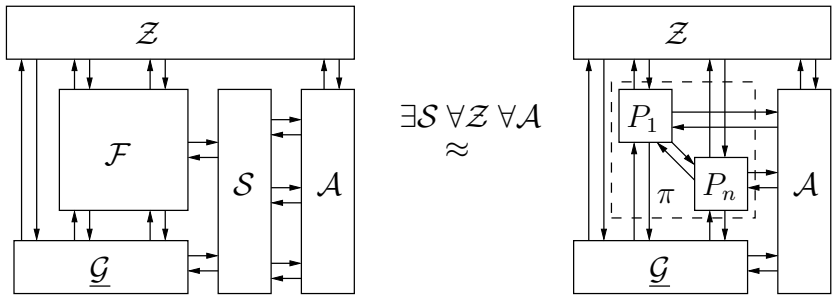


Figure 2.4: The protocol π $\underline{\mathcal{G}}$-EUC-realizes the ideal functionality $\mathcal{F}$

number of instances of $\underline{\mathcal{G}}$ running externally. It is usually more convenient to study the protocols under $\underline{\mathcal{G}}$-*externally constrained* environments. This means that the environment is allowed to execute only a single instance of $\underline{\mathcal{G}}$, shared by the protocol that the environment is distinguishing. Such a model is called EUC (externalized). It has been shown in [22] that $\underline{\mathcal{G}}$-EUC emulation implies GUC emulation for any protocol that uses only $\underline{\mathcal{G}}$ as a shared functionality.

Recall that, in the UC model, we say that a protocol π UC-*realizes* the ideal functionality $\mathcal{F}$ if it emulates a protocol ϕ where the parties are just forwarding messages between $\mathcal{Z}$ and $\mathcal{F}$. When this notion is extended to $\underline{\mathcal{G}}$-EUC, then the shared functionality $\underline{\mathcal{G}}$ is formally present also in the ideal execution (see [22]). The dummy parties are able to forward messages between $\mathcal{Z}$ and $\underline{\mathcal{G}}$, but it is not stated explicitly whether the dummy parties are able to forward messages from $\mathcal{F}$ to $\underline{\mathcal{G}}$, and it is not important for the public key infrastructure or the common reference string examples, as the parties only read data from the shared functionalities, but do not write anything to them.

In this thesis, we are interested in proving that the impact of the real protocol π on $\underline{\mathcal{G}}$ is indistinguishable from the impact of the ideal functionality $\mathcal{F}$ on $\underline{\mathcal{G}}$. In order to be able to define such an impact, we need to allow communication between $\mathcal{F}$ and $\underline{\mathcal{G}}$. Formally, we modify the definition of the $\underline{\mathcal{G}}$-EUC realization, allowing to forward messages from $\mathcal{F}$ to $\underline{\mathcal{G}}$. The real protocol π will have to enforce such forwarding in order to $\underline{\mathcal{G}}$-EUC-realize $\mathcal{F}$. Any message m that is supposed to be delivered to a shared functionality $\underline{\mathcal{G}}$ will be denoted $\underline{m}$, so that it would be clear whether the dummy parties forward this message to $\mathcal{Z}$ or to $\underline{\mathcal{G}}$. The message sent by $\mathcal{F}$ to the output port of the party P_k will arrive at $\underline{\mathcal{G}}$ from the input port of the party P_k. The pictorial representation of $\underline{\mathcal{G}}$-EUC realization is given in Figure 2.4.

2.2.4 Languages for Secure Computation

SMC applications are often written in a high level language that allows the programmer to use a set of basic operations as building blocks, without taking into

account the underlying cryptography. If all the basic protocols (called *arithmetic blackbox protocols*) are proven to be secure in the UC model, then the programmer may compose them in his applications in an arbitrary way, without needing distinct security proofs for each particular application. In this thesis, we call such applications *privacy-preserving*.

The high level languages are usually provided for existing computational platforms [106, 17, 75, 28]. These may be either domain-specific languages [15, 75, 97] or variants of general-purpose languages [39]. A program looks very similar to an ordinary imperative language (such as Java, Python, or C), but it does much more, as it is being compiled to a sequence of cryptographic protocols.

The control flow of such programs should not leak any information about the private values, so it has to be independent of them. Hence, branching on private values is often disallowed and, in general, one needs to compute all the branches simultaneously to avoid leaking information about the chosen branch. In this thesis, the program conditionals that make a choice depending on a value of a private variable are called *private conditionals*. As a part of this thesis, we provide an optimization for reducing their overhead.

2.3 Basics and Notation

In this section we give some basic notions that we use in building SMC. The notation used throughout this thesis is given in Table 2.1.

For measuring protocol cost, we take into account the number of rounds as well as the total number of bits communicated through the network. Formally, we define a type $Cost = \mathbb{N} \times \mathbb{N}$, where the first component is the bit communication, and the second component is the number of rounds. In order to handle the cost of protocol compositions more easily, we introduce the operations $\otimes : Cost \times Cost \to Cost$ (parallel composition) and $\oplus : Cost \times Cost \to Cost$ (sequential composition), defined as follows:

- $(a, b) \otimes (c, d) = (a + c, \max(b, d))$;
- $(a, b) \oplus (c, d) = (a + c, b + d)$.

We will use the shorthand $(a, b)^{\otimes n}$ to denote $(a, b) \otimes \cdots \otimes (a, b)$, where (a, b) occurs n times. Let the operation $\otimes$ have higher priority than $\oplus$.

2.3.1 Types of Indistinguishability

Let x be a value observed by the attacker who wants to extract some information out of x. In general, the ability of the attacker to get information can be reduced to a sequence of guesses, whether x comes from the distribution $\mathcal{X}_0$ or $\mathcal{X}_1$, each guess

Table 2.1: Notation

Notation	Explanation
1^n	$\underbrace{1,\ldots,1}_{n}$
$\mathsf{negl}(\eta)$	a function negligible in η
$x \leftarrow y$	assigning to a variable x the value y
$e[x \leftarrow y]$	substituting x with y in the expression e
$x \xleftarrow{\$} X$	sampling x from the uniform distribution over X
$\vec{x}$, $[x_1, \ldots, x_n]$ x_i	a vector the i-th entry of $\vec{x}$
A a_{ij}	a matrix the entry in the i-th row and j-th column of A
S_n	the group of permutations of length n
$\langle \vec{x}, \vec{y} \rangle$	$\sum_{i=1}^{\lvert\vec{x}\rvert=\lvert\vec{y}\rvert} x_i \cdot y_i$, the dot product of $\vec{x}$ and $\vec{y}$
$\vec{x} + \vec{y}$, $\alpha\vec{x}$	sum of two vectors, multiplicaton by a scalar α
$\vec{x}\ op\ \vec{y}$, $op \in \{<, \leq, =\}$	pointwise comparison of $\vec{x}$ and $\vec{y}$
$x \| y$	concatenation of vectors/bitstrings x and y
$x \circ y$	composition of x and y (for various types of x, y)
$A \subseteq B$	the set A is a subset of B
$A \subset B$	the set A is a non-inclusive subset of B
$\mathbb{N}$	the set of natural numbers
$\mathbb{Z}$	the set of integers
$\mathbb{R}$	the set of real numbers
$\mathbb{F}$	a finite field
id_X	identity function $X \to X$
$\mathsf{Dom} f$, $\mathsf{Ran} f$	domain and range of a function f
$\mathbf{Pr}[A]$	probability of the event A
$[n]$	$\{1, \ldots, n\}$
$(x_i)_{i \in [n]}$	$[x_1, \ldots, x_n]$
$[\![x]\!] = (x^k)_{k \in [n]}$	sharing of x (the scheme is implied by the context)
$[\![x]\!] := \mathsf{classify}(x)$	decomposition of x to shares
$x := \mathsf{declassify}([\![x]\!])$	reconstruction of x from shares
$[\![p]\!]$	semantics of p (used only in Chapter 6, where sharing notation $[\![\cdot]\!]$ is no longer used)
$b\ ?\ x\ :\ y$	if b then x else y
$C^{\otimes n}$, $C \otimes D$	parallel composition of protocol costs C, D
$C \oplus D$	sequential composition of protocol costs C, D

extracting one bit of information. For example, given a ciphertext x, the attacker wants to know whether the encrypted message is "Attack at night" (the distribution of possible ciphertexts is $\mathcal{X}_1$), or "Do not attack at night" (the distribution of possible ciphertexts is $\mathcal{X}_0$).

We formalize the ability of the attacker to distinguish between the distributions $\mathcal{X}_0(\eta)$ and $\mathcal{X}_1(\eta)$ parametrized by the security parameter $\eta \in \mathbb{N}$. The attacker can be represented by an algorithm $\mathcal{A}$ taking x as input. It outputs 0 if it thinks that x has come from $\mathcal{X}_0(\eta)$, and it outputs 1 if it thinks that x comes from $\mathcal{X}_1(\eta)$.

A function $f(\eta)$ called *negligible in* η if it grows slower than any inverse polynomial of η. Throughout this thesis, we use $\mathsf{negl}(\eta)$ to denote such a function. An example of negligible function is $2^{-\eta}$.

There exist the following standard notions of indistinguishability:

- *Perfect*: $|Pr_{x\leftarrow\mathcal{X}_0(\eta)}[\mathcal{A}(x) = 1] - Pr_{x\leftarrow\mathcal{X}_1(\eta)}[\mathcal{A}(x) = 1]| = 0$ for any η, i.e. the attacker cannot infer any information from x, and may only make a random guess.

- *Statistical*: $|Pr_{x\leftarrow\mathcal{X}_0(\eta)}[\mathcal{A}(x) = 1] - Pr_{x\leftarrow\mathcal{X}_1(\eta)}[\mathcal{A}(x) = 1]| \leq \mathsf{negl}(\eta)$. The attacker is able to distinguish the two sets, but his advantage can be made very small by increasing η.

- *Computational*: $|Pr_{x\leftarrow\mathcal{X}_0(\eta)}[\mathcal{A}(x) = 1] - Pr_{x\leftarrow\mathcal{X}_1(\eta)}[\mathcal{A}(x) = 1]| \leq \mathsf{negl}(\eta)$ for any $\mathcal{A}$ that works in time that is *polynomial in* η. In other words, $\mathcal{A}$ is able to distinguish $\mathcal{X}_0(\eta)$ and $\mathcal{X}_1(\eta)$ with probability higher than $\mathsf{negl}(\eta)$ only if its work time is superpolynomial in η (i.e it grows faster than any polynomial of η).

The notion of indistinguishability can be extended to protocols. We give a general definition of a δ-private protocol that covers all types of indistinguishability, depending on the chosen parameters. The idea of this definition is that, whatever inputs the corrupted parties choose, as far as all parties follow the protocol, the adversary controlling the corrupted parties cannot distinguish the protocol running on real inputs of honest parties from a protocol where all messages are simulated from the inputs of corrupted parties.

Definition 2.2 (δ-private protocol [14])**.** Let Π be a multiparty protocol that takes input $\vec{x}$ from honest parties and $\vec{y}$ from corrupted parties. Let $\Pi(\vec{x}, \vec{y})$ be a function that outputs messages received by corrupted parties during the execution of Π on inputs $\vec{x}$ and $\vec{y}$. Let A^O denote an adversary that has a blackbox access to a function O. We say that the protocol Π is *δ-private against a class of adversaries* $\mathcal{A}$ if there exists a simulator Sim, such that for all adversaries $A \in \mathcal{A}$ and inputs $\vec{x}, \vec{y}$, $\left|\mathbf{Pr}\left[A^{\Pi(\vec{x},\vec{y})}(\vec{y}) = 1\right] - \mathbf{Pr}\left[A^{\mathsf{Sim}(\vec{y})}(\vec{y}) = 1\right]\right| \leq \delta$.

2.3.2 Digital Signatures

Signatures allow to bind a message to a signer S in such a way that any other party may verify that that the message indeed originates from S and has not been modified in any way.

Definition 2.3 (Digital signature scheme [52, Definition 12.1])**.** A *signature scheme* is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ satisfying the following:

1. The key generation algorithm Gen takes 1^η as input and outputs a public/secret key pair (pk, sk) of length at least η.
2. The signing algorithm Sign takes as input a secret key sk and a message $m \in \{0,1\}^*$, outputting a signature $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$.
3. The deterministic verification algorithm Vrfy takes as input a public key pk, a message m, and a signature σ, outputting a bit $b \leftarrow \mathsf{Vrfy}_{pk}(m, \sigma)$.

It is required that, for all η, for all (pk, sk) output by $\mathsf{Gen}(1^\eta)$, and all $m \in \{0,1\}^*$, it holds that $\mathsf{Vrfy}_{pk}(m, \mathsf{Sign}_{sk}(m)) = 1$.

It should be computationally hard for the attacker to come up with a valid message/signature pair without knowing the signing key. Even if the attacker has seen some valid message/signature pairs, as far as their number is polynomial in η, it should not help him to come up with another valid message/signature pair.

Definition 2.4 (Existentially unforgeable signature scheme [52, Definition 12.2])**.** Let $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme. Let the adversary $\mathcal{A}$ conduct the following security experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(\eta)$:

1. $\mathsf{Gen}(1^\eta)$ is run to obtain keys (pk, sk).
2. The adversary $\mathcal{A}$ is given pk and access to $\mathsf{Sign}_{sk}(\cdot)$ as a black box. It requests a set of signatures $\sigma' \leftarrow \mathsf{Sign}_{sk}(m')$ for some messages m' chosen by $\mathcal{A}$ itself, where the number of requests is polynomial in η. In the end, $\mathcal{A}$ outputs (m, σ), where $m \neq m'$ for all messages m' whose signatures $\mathcal{A}$ has already requested.
3. The output if the experiment is defined to be 1 iff $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$.

The signature scheme Π is *existentially unforgeable under an adaptive chosen-message attack* if for all PPT adversaries $\mathcal{A}$ it holds that

$$\mathbf{Pr}[\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(\eta) = 1] \leq \mathsf{negl}(\eta) \ .$$

Throughout this thesis, we assume that all used signature schemes are existentially unforgeable under an adaptive chosen-message attack. In our practical experiments, we will use hashed RSA based on SHA-256, although its existential unforgeability is not formally proven [52, Section 12.3].

2.3.3 Message Authentication Codes

In multiparty protocols, it is usually sufficient for authentication that only the receiver is convinced that the message has originated from the sender S. In this case, public key cryptography is not necessary, and it suffices that each pair of parties holds a common symmetric key.

Definition 2.5 (Message authentication code (MAC) [52, Definition 4.1])**.** A *message authentication code* is a tuple of PPT algorithms $(\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ satisfying the following:

1. The key generation algorithm Gen takes 1^η as input, and outputs a key k of length at least η.
2. The tag generation algorithm Mac takes as input a key k and a message $m \in \{0,1\}^*$, outputting a tag $t \leftarrow \mathsf{Mac}_k(m)$.
3. The deterministic verification algorithm Vrfy takes as input the key k, a message m, and a tag t, outputting a bit $b \leftarrow \mathsf{Vrfy}_k(m, \sigma)$.

It is required that, for all η, for all k output by $\mathsf{Gen}(1^\eta)$, and all $m \in \{0,1\}^*$, it holds that $\mathsf{Vrfy}_k(m, \mathsf{Mac}_k(m)) = 1$.

It should be computationally hard for the attacker to come up with a valid message/tag pair without knowing the key. Even if the attacker has seen some valid message/tag pairs, as far as their number is polynomial in η, it should not help him to come up with another valid message/tag pair.

Definition 2.6 (Existentially unforgeabile MAC [52, Definition 4.2])**.** Let the tuple $\Pi = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a message authentication code. Let the adversary $\mathcal{A}$ conduct the following security experiment $\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(\eta)$:

1. $\mathsf{Gen}(1^\eta)$ is run to obtain a key k.
2. The adversary $\mathcal{A}$ is given 1^η as input, and access to $\mathsf{Mac}_k(\cdot)$ as a black box. It requests a set of tags $t \leftarrow \mathsf{Mac}_k(m)$ for some set $m \in Q$, where the number of requests is polynomial in η. In the end, $\mathcal{A}$ outputs (m, t).
3. The output of the experiment is defined to be 1 iff $\mathsf{Vrfy}_k(m, t) = 1$.

The MAC Π is *existentially unforgeable under an adaptive chosen-message attack* if for all PPT adversaries $\mathcal{A}$ it holds that

$$\mathbf{Pr}[\mathsf{Mac\text{-}forge}_{\mathcal{A},\Pi}(\eta) = 1] \leq \mathsf{negl}(\eta) \ .$$

2.3.4 Hash Functions and Merkle Tree

A hash function is a function that maps inputs coming from a large (possibly infinite) set to some smaller set of fixed size, trying to keep the number of collisions small. In cryptographic applications, it is often required that not just the number of collisions is small, but it is computationally hard for the adversary to find a collision.

Definition 2.7 (Hash function [52, Definition 4.11])**.** A *hash function* is a pair of PPT algorithms (Gen, H) satisfying the following:

1. Gen takes as input a security parameter 1^η and outputs a key s.
2. There exists a polynomial ℓ such that H takes as input a key s and a bitstring $x \in \{0,1^*\}$, and outputs a bitstring $H^s(x) \in \{0,1\}^{\ell(\eta)}$.

It should be computationally hard for the attacker to come up with two inputs $x \neq x'$ such that $H(x) = H(x')$.

Definition 2.8 (Collision resistant hash function [52, Definition 4.12])**.** Let $\Pi = (\mathsf{Gen}, H)$ be a hash function. Let the adversary $\mathcal{A}$ conduct the following security experiment $\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(\eta)$:

1. $\mathsf{Gen}(1^\eta)$ is run to obtain a key s.
2. The adversary $\mathcal{A}$ is given s. It outputs x, x'.
3. The output of the experiment is defined to be 1 iff $x \neq x'$ and $H^s(x) = H^s(x')$.

The hash function Π is *collision resistant* if for all PPT adversaries $\mathcal{A}$ it holds that

$$\mathbf{Pr}[\mathsf{Hash\text{-}coll}_{\mathcal{A},\Pi}(\eta) = 1] \leq \mathsf{negl}(\eta) \ .$$

Throughout this paper, we will omit the explicit key s from the notation, and denote a hash function by H.

Merkle hash tree [78]. Hash functions can be used to verify signatures of single messages that are signed in batches. Let $m_1, \ldots, m_n$ (for simplicity, let n be a power of two) be the messages for which we want to generate just one signature. Suppose that we want to get a signature scheme that only allows to check whether m_i was signed correctly, without opening any other m_j. Given a collision-resistant hash function H, the signer first computes $h_i = H(m_i \| r_i)$, where r_i is freshly generated randomness. It then partitions the values h_i into pairs, and computes $h_{ij} = H(h_i \| h_j \| r_{ij})$, where r_{ij} is again some fresh randomness. Applying this

step recursively to h_{ij}, treating them similarly to m_i, it constructs a binary tree of height $\log n$, getting a single hash h that it eventually signs. At any time when the signature should be verified on m_i, it is sufficient to open the randomness r_i and the siblings of all hashes in the binary tree on the way from m_i to the root (there are $\log n$ of them). Using new randomness in each hash ensures that the siblings do not leak any information about the other messages m_j.

2.3.5 Finite Fields and the Schwarz-Zippel Lemma

In our protocols, we will often need to check whether some quantity equals 0. In general, there is a multiset X of values, and for all $x \in X$ we want to check whether $x = 0$. In the SMC setting, comparison is not an easy operation, and we would like to make the number of comparisons much smaller than $|X|$. If $X \subseteq \mathbb{F}$, then we may make use of well-known results from field theory.

Lemma 2.1 (Schwarz-Zippel lemma [81, Theorem 7.2])**.** *Let $\mathbb{F}$ be a finite field. Let $p(x_1, \ldots, x_n) \neq 0$ be a multivariate polynomial of degree d. Let $S \subseteq \mathbb{F}$ be any subset of $\mathbb{F}$. Let $y_1, \ldots, y_n \xleftarrow{\$} S^n$. Then $\mathbf{Pr}[p(y_1, \ldots, y_n) = 0] \leq \frac{d}{|S|}$.*

In particular, instead of checking $x_i = 0$ one by one, we may generate n random numbers $r_1, \ldots, r_n \xleftarrow{\$} \mathbb{F}$, and verify $r_1 \cdot x_1 + \cdots + r_n \cdot x_n = 0$ instead. By the Schwarz-Zippel lemma, if this value equals 0, then $x_i = 0$ for all $i \in [n]$ with probability $\geq 1 - \frac{1}{|\mathbb{F}|}$, which can be increased with the size of $\mathbb{F}$. In the SMC setting, multiplication is usually more efficient than comparison, and it is significantly more efficient in the algorithms that we use in our protocols.

In our protocols, we use finite fields of the form $\mathbb{Z}_q$ for a prime number $q > 2$. We also make use of the fact that $\mathbb{Z}_q^*$ is a multiplicative group under modular multiplication.

2.3.6 Linear Programming

We will reduce some optimizations to well-known standard methods. A *linear programming* task (LP) is an optimization task stated as

$$\text{minimize } \langle \vec{c}, \vec{x} \rangle, \text{ subject to } A\vec{x} \leq \vec{b}, \vec{x} \geq \vec{0} \ , \tag{2.1}$$

where $\vec{x} \in \mathbb{R}^n$ is a vector of optimization variables, and the quantities $A \in \mathbb{R}^{m \times n}$, $\vec{b} \in \mathbb{R}^m$, $\vec{c} \in \mathbb{R}^n$ are the parameters that define the task itself.

Adding the constraint that $x_i \in \mathbb{Z}$ for $i \in \mathcal{I}$ for some $\mathcal{I} \subseteq \{1, \ldots, n\}$ gives us a *mixed integer linear programming* task (ILP).

While linear programming can be solved in time polynomial in n, m, this is not the case for mixed integer linear programming. However, the existing ILP solvers (e.g. [44]) are quite efficient in practice.

2.4 Linear Secret Sharing Schemes

The main idea of secret sharing is that no party is able to infer any information about the shared value, observing just its own share. Since the protocols often assume the existence of several corrupted parties, it is required that no subset of parties is able to reconstruct the secret, unless there are sufficiently many shares (e.g. all of them). We give some examples of secret sharing schemes that we will use in this thesis.

2.4.1 Additive Sharing

We start from additive sharing, which is one of the simplest examples of secret sharing. Let $x \in \mathbb{Z}_m$ be a secret that we want to share among n parties. Let $x_i \xleftarrow{\$} \mathbb{Z}_m$ for $i \in [n-1]$, and $x_n = x - x_1 - \cdots - x_{n-1}$. It is known that not only $x_1, \ldots, x_{n-1}$ are independent uniformly distributed variables, but also any subset of $n-1$ variables $x_{i_1}, \ldots, x_{i_{n-1}}$ for $\{i_1, \ldots, i_{n-1}\} = \mathcal{I} \subset [n]$ (see e.g. [16]). Therefore, seeing the values $x_{i_1}, \ldots, x_{i_{n-1}}$ does not give any clue about what x could be, since the remaining x_i for $i \in [n] \setminus \mathcal{I}$ may change the final result to any element of $\mathbb{Z}_m$ with equal probability.

In additive sharing, two shared values can be added by summing their shares, which can be done locally by each party. Multiplication requires some interaction between the parties. For three parties, it can be done quite easily using for example the protocol of [17]. Defining addition and multiplication is sufficient to perform any computation, although more efficient elaborated protocols can be developed.

The security properties hold not only in $\mathbb{Z}_m$ using the addition operation $(+)$, but also in any group $(A, \oplus)$ using the group operation $\oplus$. We are now ready to give a more general definition of linear secret sharing. Let $(R, +, \cdot)$ be a ring. A sharing scheme is called linear if the shared secret $x \in R$ can be viewed as a linear combination of its shares $x_1, \ldots, x_n \in R$, i.e. $x = \alpha_1 \cdot x_1 + \cdots + \alpha_n \cdot x_n$ for some constants $\alpha_i \in A \subseteq R$. By reasoning similar to additive sharing, sampling $x_1, \ldots, x_{n-1} \xleftarrow{\$} R$, and taking $x_n = \alpha_n^{-1} \cdot (x - \sum_{i=1}^{n-1} \alpha_i \cdot x_i)$ should keep any subset of $n-1$ shares uniformly distributed to the observer. The choice of the set A depends on the type of R, e.g. $A = R \setminus \{0\}$ is suitable for fields, and $A = \{1\}$ for arbitrary rings.

2.4.2 Linear Threshold Sharing

It is often useful to share a secret among n parties in such a way that any t parties are able to recover it, but fewer than t are not. For example, a government consisting of n members may be officially allowed to use a dangerous weapon only if at least t of them agree on it. The weapon could be locked in a safe, and the key distributed

among the n members in such a way that at least t shares are required to open the lock. Instead of a weapon, the safe could contain any confidential information that is only allowed to be opened or involved into computation if sufficiently many parties agree on it. This kind of sharing is called (n, t)-threshold secret sharing.

Replicated secret sharing [25]. An (n, t)-threshold sharing for arbitrary rings can be constructed on the basis of additive sharing. Let $a \in R$ for some finite ring R. Let $\mathcal{V}_1, \ldots, \mathcal{V}_{\binom{n}{t}}$ be all subsets of $[n]$ of size t. The share of each party P_k is a vector $\vec{a}^k \in R^{\binom{n}{t}}$, such that for each $j \in [\binom{n}{t}]$, the equation $\sum_{k \in \mathcal{V}_j} a_j^k = a$ holds. Also, $a_j^k = 0$ whenever $k \notin \mathcal{V}_j$. In other words, the same value a is additively shared in $\binom{n}{t}$ different ways, each time issuing some shares $a_1, \ldots, a_t$ such that $a_1 + \ldots + a_t = a$ to a certain subset of t parties. All these $\binom{n}{t}$ sharings are independent. In this way, any t parties are able to reconstruct the secret. However, to any set of less than t parties, the shares should look uniformly distributed and independent.

Shamir's secret sharing [99]. Replicated secret sharing scales badly with n. For finite fields, more efficient (n, t)-threshold sharings are available. Let $\mathbb{F}$ be a finite field. Shamir's secret sharing is defined as follows:

- Before the computation, the parties agree on n distinct field elements $\alpha_i \in \mathbb{F} \setminus \{0, 1\}$ for $i \in \{1, \ldots, n\}$. All these values are public.
- Let $f_0 \in \mathbb{F}$ be a value that some party P_k wants to share. P_k generates independent random values $f_1, \ldots, f_{t-1} \xleftarrow{\$} \mathbb{F}$. Let $F(x)$ be the polynomial $f_0 + f_1 x + f_2 x^2 + \cdots + f_{t-1} x^{t-1}$. P_k evaluates $F(\alpha_i)$ for all $i \in [n]$, and for $i \neq k$ it sends $F(\alpha_i)$ to P_i. In matrix terms, this operation can be represented in the following way.

$$\begin{pmatrix} F(\alpha_1) \\ F(\alpha_2) \\ \vdots \\ F(\alpha_n) \end{pmatrix} = \begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^{t-1} \\ 1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^{t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \alpha_n^2 & \ldots & \alpha_n^{t-1} \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{t-1} \end{pmatrix}$$

- Let $T = \{i_1, \ldots, i_t\}$ be an arbitrary subset of t parties that want to reconstruct the secret. They need to solve the following equation system.

$$\begin{pmatrix} 1 & \alpha_{i_1} & \alpha_{i_1}^2 & \ldots & \alpha_{i_1}^{t-1} \\ 1 & \alpha_{i_2} & \alpha_{i_2}^2 & \ldots & \alpha_{i_2}^{t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_{i_t} & \alpha_{i_t}^2 & \ldots & \alpha_{i_t}^{t-1} \end{pmatrix} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{t-1} \end{pmatrix} = \begin{pmatrix} F(\alpha_{i_1}) \\ F(\alpha_{i_2}) \\ \vdots \\ F(\alpha_{i_t}) \end{pmatrix}$$

Let A_T denote the matrix that corresponds to α_i for $i \in T$. It turns out that any matrix of such form (a *Vandermonde* matrix [81, Definition 7.1]) is invertible. Hence the t parties may compute

$$\begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{t-1} \end{pmatrix} = A_T^{-1} \begin{pmatrix} F(\alpha_{i_1}) \\ F(\alpha_{i_2}) \\ \vdots \\ F(\alpha_{i_t}) \end{pmatrix} \ .$$

Since the parties are interested only in f_0, it is sufficient to compute just one scalar product. Let $\vec{b}$ denote the first row of A_T^{-1} The parties have to compute

$$f_0 = b_1 F(\alpha_{i_1}) + \cdots + b_t F(\alpha_{i_t}) \ .$$

Similarly to additive sharing, the addition of values can be done locally by adding the shares, and multiplication requires some interaction. It can be done using e.g. the protocol of [26].

2.4.3 Permutation Sharing

Let S_m be the group of permutations of length m. It is known that S_m is a group w.r.t. the composition operation $(\circ)$. Hence, we can apply the idea of additive sharing to secret share permutations as $\pi = \pi_1 \circ \cdots \circ \pi_n$. Let $\pi_i \xleftarrow{\$} S_m$ for $i \in [n-1]$, and $\pi_n = \pi_{n-1}^{-1} \circ \ldots \circ \pi_1^{-1} \circ \pi$. Seeing $\pi_{i_1}, \ldots, \pi_{i_{n-1}}$ for $\{i_1, \ldots, i_{n-1}\} = \mathcal{I} \subset [n]$ does not give any clue about what π could be, since π_i for $i \in [n] \setminus \mathcal{I}$ may change the final result to any element of S_m with equal probability. A threshold variant of permutation sharing can be obtained using the idea of replicated sharing, letting the same permutation be shared in $\binom{n}{t}$ ways.

2.5 Correlated Randomness

It is often useful to let the parties share some correlated randomness. In particular, there is a uniformly distributed random bitstring s_x, and additionally $s_y \leftarrow f(s_x)$ for some certain function f. While a uniformly distributed s_x is relatively easy to generate bit by bit using any coin toss protocol (see e.g. [24]), computing s_y is in general non-trivial.

A simple example of correlated randomness is Beaver multiplication triple [8]. These are triples of values (a, b, c) in some ring $\mathbb{Z}_n$, such that $a, b \xleftarrow{\$} \mathbb{Z}_n$ are distributed uniformly, and $c = a \cdot b$. Precomputing such triples can be used to linearize multiplications. For example, if we want to multiply $x \cdot y$, and there is a triple (r_x, r_y, r_{xy}) already precomputed and preshared, we may first compute and

publish $\hat{x} := x - r_x$ and $\hat{y} := y - r_y$, and then compute the linear combination $x \cdot y = (\hat{x} + r_x)(\hat{y} + r_y) = \hat{x}\hat{y} + r_x\hat{y} + \hat{x}r_y + r_x r_y = \hat{x}\hat{y} + r_x\hat{y} + \hat{x}r_y + r_{xy}$. Publishing $\hat{x}$ and $\hat{y}$ leaks no information about x and y, since they are masked by uniformly distributed r_x and r_y. This works similarly to hiding the secrets in additive sharing (Section 2.4).

Applying Beaver triples yields another way to implement the multiplication protocol for linear secret sharing. Although the online protocol phase becomes cheaper, all the multiplications are just being pushed into the preprocessing phase in which the triples are generated, and hence the total amount of communication between the parties may only increase. A good property of this approach is that the generation of all triples for the protocol can be done in parallel, so the total running time may actually decrease.

2.6 Commitments

In this thesis, we will verify the correctness of parties' computation w.r.t. their inputs, outputs, and the messages that they exchanged. In order to make this possible, each party must be somehow bound to all these quantities. For this, we will need to use *commitments*. First, let us give a general definition of a commitment scheme.

Definition 2.9 (Commitment scheme [45, Definition 4.4.1], extended to arbitrary messages)**.** A *commitment scheme* is a tuple of PPT algorithms $(\mathsf{Gen}, \mathsf{Comm}, \mathsf{Open})$ satisfying the following:

1. Gen takes as input a security parameter 1^η, and outputs a public key pk.
2. The commitment algorithm Comm takes as input the key pk and a message $m \in \{0,1\}^*$, outputting a pair $(c, d) \leftarrow \mathsf{Comm}_k(m)$. The value c is public to everyone, while d is initially known only to the party committed to m.
3. The deterministic verification algorithm Open takes as input the key k, the pair (c, d), and outputs $m \leftarrow \mathsf{Open}_{pk}(c, d)$. It is possible that $m = \perp$.

It is required that, for all η, for all pk output by $\mathsf{Gen}(1^\eta)$, and all $m \in \{0,1\}^*$, it holds that $\mathsf{Open}_{pk}(\mathsf{Comm}_k(m)) = m$.

The idea behind commitment schemes is that the value c should not leak any information about the committed m. The message m is opened by publishing d, which in general happens at some point after some other protocol interactions have taken place. However, publishing c should ensure that the committing party (the *committer*) is bound to m, and it cannot later choose a different $d' \neq d$ to open a message $m' \neq m$. These requirements are expressed by the following security definitions.

Definition 2.10 (Binding property [45, Definition 4.4.1], extended to arbitrary messages)**.** Let $\Pi = (\mathsf{Gen}, \mathsf{Comm}, \mathsf{Open})$ be a commitment scheme. Let the adversary $\mathcal{A}$ conduct the following security experiment $\mathsf{Comm\text{-}bind}_{\mathcal{A},\Pi}(\eta)$:

1. $\mathsf{Gen}(1^\eta)$ is run to obtain a public key pk.
2. The adversary $\mathcal{A}$ is given pk. It outputs c, d_0, d_1.
3. The output if the experiment is defined to be 1 iff $m_0 \neq m_1$, $m_0 \neq \perp$, $m_1 \neq \perp$ for $m_0 \leftarrow \mathsf{Open}_{pk}(c, d_0)$ and $m_0 \leftarrow \mathsf{Open}_{pk}(c, d_1)$.

A commitment scheme $\Pi = (\mathsf{Gen}, \mathsf{Comm}, \mathsf{Open})$ is *binding* if for all PPT adversaries $\mathcal{A}$ it holds that

$$\mathbf{Pr}[\mathsf{Comm\text{-}bind}_{\mathcal{A},\Pi}(\eta) = 1] \leq \mathsf{negl}(\eta) \ .$$

Definition 2.11 (Hiding property [45, Definition 4.4.1], extended to arbitrary messages)**.** A commitment scheme $\Pi = (\mathsf{Gen}, \mathsf{Comm}, \mathsf{Open})$ is *hiding* if for all PPT adversaries $\mathcal{A}$, for any pair of messages (m_0, m_1) chosen by $\mathcal{A}$, $pk \leftarrow \mathsf{Gen}(1^\eta)$, it holds that

$$|\mathbf{Pr}[\mathcal{A}(\mathsf{Comm}_{pk}(m_0)) = 1] - \mathbf{Pr}[\mathcal{A}(\mathsf{Comm}_{pk}(m_1)) = 1]| \leq \mathsf{negl}(\eta) \ .$$

In this thesis, we will use a particular kind of commitment based on linear (n, t)-threshold secret sharing (defined in Section 2.4) for $t = \lceil n/2 \rceil + 1$. This kind of commitment works under the *honest majority* assumption. The public key pk can be viewed as a set of parameters defining a certain sharing scheme (e.g. the coefficients α_i of Shamir's sharing). Without loss of generality, let the message m be a ring or a field element, depending on the used sharing scheme. The algorithm Comm shares the message m among the n parties, where each share is signed by the committer. Each other party may treat the share it receives as c, and the remaining $n - 1$ shares as d. The algorithm Open reconstructs m from all the shares that are provided with valid signatures of the committer. If the resulting shares are inconsistent, then Open outputs $\perp$. Since there are at least t honest parties, and t shares are sufficient to reconstruct the secret, $\mathsf{Open}_{pk}(\mathsf{Comm}_k(m)) = m$ always holds for an honest committer that never signs inconsistent shares.

This commitment scheme is perfectly hiding since no set of less than t parties is able to reconstruct the shared value, and by assumption there are strictly less than t corrupted parties. It is perfectly binding since tampering with the shares of corrupted parties may only lead to inconsistency of shares, causing $\perp$ to be output. Such a commitment is *homomorphic*: linear secret sharing allows to compute any linear combinations of commitments without opening them. Since the signatures of shares are not homomorphic, opening the linear combinations is a bit more tricky. We will discuss it in more detail in Chapter 4.

2.7 Verifiable Computation

In practical settings, it is often sufficient that the active adversary is detected not immediately after the malicious act, but at some point later. Hence, ideas from verifiable computation (VC) [43] are applicable to SMC. In general, VC allows a weak client to outsource a computation to a more powerful server that accompanies the computed result with a proof of correct computation that is relatively easy for the weak client to verify. Similar ideas can be used to strengthen protocols secure against passive adversaries: after execution, each party will prove to others that it has correctly followed the protocol.

2.7.1 Linear Probabilistically Checkable Proofs

First, we give an intuitive description of what an interactive proof is. Let $\mathcal{L}$ be a formal language defined over bitstrings. For the given bitstring x, the goal of the party P (the prover) is to convince the party V (the verifier) that $x \in \mathcal{L}$. Let $\mathcal{R}$ be a binary relation such that $\mathcal{R}(x, w) = 1$ iff w is a *witness* proving that $x \in \mathcal{L}$. In general, proving that $x \in \mathcal{L}$ reduces to proving the existence of w s.t $\mathcal{R}(x, w) = 1$. It is possible that V does not get any information about w, just a single bit $\exists w : \mathcal{R}(x, w) = 1$, such proofs are called zero-knowledge. The prover may be allowed to cheat with some probability, such proofs are called probabilistic.

Let us now become more concrete and assume that $\mathcal{L}$ is a language of vectors over a finite field $\mathbb{F}$. We now define a narrower class of interactive proofs that we will use in this thesis.

Definition 2.12 (Linear Probabilistically Checkable Proof (LPCP) [10])**.** Let $\mathbb{F}$ be a finite field, $k, \ell \in \mathbb{N}$, $\mathcal{R} \subseteq \mathbb{F}^k \times \mathbb{F}^\ell$. Let $\mathcal{P}$ and $\mathcal{Q}$ be probabilistic algorithms, and $\mathcal{D}$ a deterministic algorithm. The pair $(\mathcal{P}, \mathcal{V})$, where $\mathcal{V} = (\mathcal{Q}, \mathcal{D})$ is a *d-query δ-statistical HVZK linear PCP for $\mathcal{R}$ with knowledge error ε and query length m*, if the following holds.

Syntax. On input $\vec{v} \in \mathbb{F}^k$ and $\vec{w} \in \mathbb{F}^\ell$, the algorithm $\mathcal{P}$ computes $\vec{\pi} \in \mathbb{F}^m$. The algorithm $\mathcal{Q}$ randomly generates d vectors $\vec{q}_1, \ldots \vec{q}_d \in \mathbb{F}^m$ and some *state information* $\vec{u}$. Let $\mathcal{V}^{\vec{\pi}}(\vec{v})$ denote the execution of $\mathcal{Q}$ followed by the execution of $\mathcal{D}$ on input $(\vec{v}, \vec{u}, a_1, \ldots, a_d)$, where $\vec{u}$ is the output of $\mathcal{Q}$, and $a_i = \langle \vec{\pi}, \vec{q}_i \rangle$. The algorithm $\mathcal{D}$ either accepts the input (outputs 1) or rejects the input (outputs 0).

Completeness. For every $(\vec{v}, \vec{w}) \in \mathcal{R}$, the output of $\mathcal{P}(\vec{v}, \vec{w})$ is a vector $\vec{\pi} \in \mathbb{F}^m$ such that $\mathbf{Pr}[\mathcal{V}^{\vec{\pi}}(\vec{v}) = 1] = 1$.

Knowledge. There exists a knowledge extractor $\mathcal{E}$ such that for all $\vec{\pi^*} \in \mathbb{F}^m$, if $\mathbf{Pr}\left[\mathcal{V}^{\vec{\pi^*}}(\vec{v}) = 1\right] \geq \varepsilon$ then $\mathcal{E}(\vec{\pi^*}, \vec{v})$ outputs $\vec{w}$ such that $(\vec{v}, \vec{w}) \in \mathcal{R}$.

Honest Verifier Zero Knowledge (HVZK). The protocol between an honest prover executing $\vec{\pi} \leftarrow \mathcal{P}(\vec{v}, \vec{w})$ and adversarial verifier executing $\mathcal{V}^{\vec{\pi}}(\vec{v})$ with common input $\vec{v}$ and prover's input $\vec{w}$ is δ-private (see Definition 2.2) for the class of passive adversaries.

In the settings of verifiable computation, prover P is the party performing the computation, and verifier V the party that needs to be convinced that the computation was done correctly. In this case, $\vec{v}$ can be viewed as a commitment on all the prover's inputs and outputs. The vector $\vec{w}$ consists of all values known to the prover, that it uses to construct the proof $\vec{\pi} = \mathcal{P}(\vec{v}, \vec{w})$. The proof $\vec{\pi}$ consists of some helpful hints from the prover, convincing the verifier that the prover knows $\vec{w}$ such that $(\vec{v}, \vec{w}) \in \mathcal{R}$. Given access to $\vec{\pi}$, the verifier V runs the algorithm $\mathcal{V}^{\vec{\pi}}(\vec{v})$ to check whether $\vec{\pi}$ proves that the computation was correct w.r.t. $\vec{w}$. Note that V is not given a direct access to π, but calls $\mathcal{V}^{\vec{\pi}}(\cdot)$ as a black box. If it outputs 1, the proof of P is believable due to the knowledge property. Any proof of an honest P will be accepted due to the completeness property. If V follows the protocol, no information about potentially private $\vec{w}$ is leaked to V due to the HVZK property.

2.7.2 Verification as Quadratic Arithmetic Program

We now describe a particular verification scheme proposed in [9]. The main ideas behind the verification mechanism are not the main contribution of [9], and there are many other works [10, 72, 9, 85, 98] that are similar to it in their nature.

For simplicity, let us assume that the computation of the prover is represented by an arithmetic circuit. Given committed inputs and outputs of the circuit, the task of the prover is to convince the verifier that there exist valuations of intermediate gates, and possibly some additional inputs, such that the circuit indeed produces the given output with the given input. Let each gate input wire and each gate output wire be represented by a variable. Then we may rewrite the computation of each gate as an equation. For example, an addition gate can be written as $x + y = z$, where x and y correspond to the inputs and z to to the output. If the output is in turn used as an input for some other gate, we just reuse the same variable.

A *multiplication subcircuit* is an arithmetic circuit that has exactly one two-variable multiplication gate, which is the one that outputs the final result. Each arithmetic circuit can be seen as a composition of such subcircuits.

Verifying multiplication subcircuits. Let $\mathcal{C}$ be a multiplication subcircuit with input $\vec{x} = [x_1, \ldots, x_m]$ and output $\vec{y} = [y]$. The function it computes can be written as

$$y = \rho_1(x_1, \ldots, x_m) \cdot \rho_2(x_1, \ldots, x_m)$$

for some linear functions ρ_1, ρ_2, i.e. there is a vector of constants $\vec{d_i}$ such that $\rho_i(x_1, \ldots, x_m) = \langle \vec{d_i}, [1] \| \vec{x} \rangle$. In this case, the equality is of the form

$$\langle [0, \ldots, 0, 1], \vec{w} \rangle = \langle \vec{d_1} \| [0], \vec{w} \rangle \cdot \langle \vec{d_2} \| [0], \vec{w} \rangle$$

for $\vec{w} = [1] \| \vec{x} \| \vec{y}$. This can be viewed as $A\vec{w} \cdot B\vec{w} = C\vec{w}$ for one-row matrices $A = [\vec{d_1} || [0]]$, $B = [\vec{d_2} || [0]]$, and $C = [[0, \ldots, 0, 1]]$, where $(\cdot)$ denotes pointwise vector multiplication.

Verifying subcircuit composition. Let (A_1, B_1, C_1) and (A_2, B_2, C_2) be two triples of matrices for verification of subcircuits $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively. Let some inputs of $\mathcal{C}_2$ be some outputs of $\mathcal{C}_1$. Let I_1 be the variables used by $\mathcal{C}_1$, and I_2 the variables used by $\mathcal{C}_2$. In general, it is possible that $I_1 \cap I_2 \neq \emptyset$. Each variable in I_i corresponds to a certain column of each of the matrices A_i, B_i, C_i. We can extend A_i, B_i, C_i to the entire vector of variables $I_1 \cup I_2$ by introducing zero columns for the unused variables, obtaining the matrices A_i', B_i', C_i' of $|I_1 \cup I_2|$ columns, where, again, each variable is represented by exactly one column. The composition (A, B, C) of (A_1, B_1, C_1) and (A_2, B_2, C_2) includes all the checks that (A_1, B_1, C_1) and (A_2, B_2, C_2) do. It is defined as

$$A := \begin{pmatrix} A_1' \\ A_2' \end{pmatrix}, B := \begin{pmatrix} B_1' \\ B_2' \end{pmatrix}, C := \begin{pmatrix} C_1' \\ C_2' \end{pmatrix} \quad .$$

In this way, each multiplication gate of the circuit contributes a row to A, B, C, and each wire contributes a column. In addition to the initial input and output vectors $\vec{x}$ and $\vec{y}$, there will be some new variables $\vec{z}$ that represent the values of the intermediate multiplication gate outputs, so in general $\vec{w} = [1] \| \vec{x} \| \vec{y} \| \vec{z}$. If the variables come in a different order, we may always reorder the columns of A,B,C if necessary.

Example 2.1. Let $\mathcal{C}$ be a circuit computing the function $2x_1 \cdot x_2 + (x_1 + x_2) \cdot x_2 + 1$ from the inputs x_1 and x_2. Let z_1 and z_2 be intermediate variables such that $z_1 = 2x_1 \cdot x_2$ and $z_2 = (x_1 + x_2) \cdot x_2$. Let y be the final result.

The proof that y is indeed the output of $\mathcal{C}$ on the inputs x_1 and x_2 can be reduced to the proof of existence of z_1 and z_2 satisfying the following equation, where $\cdot$ denotes pointwise product of two vectors:

$$\begin{pmatrix} 0&2&0&0&0&0 \\ 0&1&1&0&0&0 \\ 1&0&0&0&1&1 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ y \\ z_1 \\ z_2 \end{pmatrix} \cdot \begin{pmatrix} 0&0&1&0&0&0 \\ 0&0&1&0&0&0 \\ 1&0&0&0&0&0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ y \\ z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \\ 0&0&0&1&0&0 \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ y \\ z_1 \\ z_2 \end{pmatrix} \quad .$$

We now formalize the properties that the matrix triple (A, B, C) must satisfy.

Definition 2.13 (Quadratic Arithmetic Program)**.** Consider integers m, n, k such that $n - 1 \geq k$. A *strong quadratic arithmetic program (*QAP*)* over a field $\mathbb{F}$, denoted $\mathcal{P}(A, B, C)$, consists of three $m \times n$ matrices A, B, C over a field $\mathbb{F}$. $\mathcal{P}$ accepts a vector $\vec{v} \in \mathbb{F}^k$ iff there exists a vector $\vec{w} = [1, w_1, \ldots, w_{n-1}]$ such that $[w_1, \ldots, w_k] = \vec{v}$ and $A\vec{w} \cdot B\vec{w} = C\vec{w}$. This defines a relation $R_{\mathcal{P}(A,B,C)}$:

$$\exists \vec{w}\ (\vec{v}, \vec{w}) \in R_{\mathcal{P}(A,B,C)} \iff \mathcal{P}(A, B, C) \text{ accepts on input } \vec{v}\ .$$

Let $\mathcal{C}$ be the arithmetic circuit, and A, B, C the matrices derived from this circuit. By construction, if $\mathcal{C}(\vec{x}) = \vec{y}$ then $\mathcal{P}(A, B, C)$ accepts $[\vec{x}||\vec{y}]$. Conversely, if $\mathcal{P}(A, B, C)$ accepts $\vec{x}\|\vec{y}$, then $\mathcal{C}(\vec{x}) = \vec{y}$. It follows from the definition that accepting means satisfying each row constraint. This in turn means that each multiplication subcircuit is computed correctly. Reusing the same variables in different rows ensures that the composition of multiplication subcircuits is taken into account.

Let N_x be the number of inputs, N_y the number of outputs, N_g the total number of gates, and $N_m \leq N_g$ the number of multiplication gates in a circuit $\mathcal{C}$. Without taking into account repetitions of intermediate forking output wires, $N_w = N_x + N_g$ is the total number of wires in $\mathcal{C}$.

Let $R_{\mathcal{P}(A,B,C)}$ be the relation for verifying $\mathcal{C}$. The relation is defined over pairs $(\vec{v}, \vec{w}) = \mathbb{F}^{N_x+N_y} \times \mathbb{F}^{N_w}$, where $\vec{v}$, and hence the first $N_x + N_y$ entries of $\vec{w}$, contain the N_x inputs and the N_y outputs. The rest $N_g - N_y$ entries of $\vec{w}$ are the intermediate values. By construction, the matrices A, B, C are all in $\mathbb{F}^{N_m \times N_w}$. We note that $N_m, N_w \in O(|\mathcal{C}|)$, i.e. they are linear in the circuit size.

2.7.3 LPCP for Quadratic Arithmetic Programs

The problem of verifying circuit computation has been reduced to the problem of proving the existence of $\vec{w}$ such that $(\vec{v}, \vec{w}) \in R_{\mathcal{P}(A,B,C)}$, where A,B,C are defined by the arithmetic circuit that the party computes, and $\vec{v}$ is the vector of inputs and outputs of the circuit. We now define a LPCP for verifying whether $\exists \vec{w}\ (\vec{v}, \vec{w}) \in R_{\mathcal{P}(A,B,C)}$. Let $A, B, C \in \mathbb{F}^{m \times n}$, $|\vec{v}| = k$.

Preprocessing: Let $A = (a_{ij})$, $B = (b_{ij})$, $C = (c_{ij})$ for $i \in \{0, \ldots, m-1\}$, $j \in \{0, \ldots, n-1\}$. Let ω be the principal m-th root of unity in $\mathbb{F}$ (we assume that $\mathbb{F}$ is chosen in such a way that this root exists).

Let A_j, B_j, C_j be polynomials of degree $m - 1$ defined in such a way that $A_j(\omega^i) = a_{ij}$, $B_j(\omega^i) = b_{ij}$, $C_j(\omega^i) = c_{ij}$. The coefficients of these polynomials can be computed for example using the Fast Fourier Transform [89]. These polynomials have degree $m - 1$ since they are defined on m distinct points.

Let $\vec{A}(x) := [A_0(x), \ldots, A_{n-1}(x)]$, $\vec{B}(x) := [B_0(x), \ldots, B_{n-1}(x)]$, $\vec{C}(x) := [C_0(x), \ldots, C_{n-1}(x)]$.

Let $S = \{\omega^0, \ldots, \omega^{m-1}\} \subseteq \mathbb{F}$. Let $Z_S(x) := \prod_{s \in S}(x - s)$ be an m-degree polynomial over $\mathbb{F}$. It has exactly m roots which are the elements of S.

The set S and the coefficients of $\vec{A}(x)$, $\vec{B}(x)$, $\vec{C}(x)$, $Z_S(x)$ are published.

Linear PCP Prover algorithm $\mathcal{P}(\vec{v}, \vec{w})$: Let $\vec{v} \in \mathbb{F}^k$, $\vec{w} \in \mathbb{F}^n$.

- Let $\delta_A, \delta_B, \delta_C \xleftarrow{\$} \mathbb{F}$ be random independent field elements.
- Let $A(x)$, $B(x)$, $C(x)$ be polynomials such that:

$$A(x) := \langle \vec{w}, \vec{A}(x) \rangle + \delta_A Z_S(x) \ ,$$

$$B(x) := \langle \vec{w}, \vec{B}(x) \rangle + \delta_B Z_S(x) \ ,$$

$$C(x) := \langle \vec{w}, \vec{C}(x) \rangle + \delta_C Z_S(x) \ .$$

  All these polynomials have degree m since the degree of each polynomial in $\vec{A}(x)$, $\vec{B}(x)$, $\vec{C}(x)$ is $m - 1$, and the degree of $Z_S(x)$ is m.

- Let $\vec{h} = [h_0, \ldots, h_m]$ be the coefficients of the polynomial

$$H(x) := \frac{A(x)B(x) - C(x)}{Z_S(x)} \ .$$

The algorithm returns $\vec{\pi} = [\delta_A, \delta_B, \delta_C] || \vec{w} || \vec{h}$. All values can be computed by the prover in time $O(|\mathcal{C}| \log |\mathcal{C}|)$. Details of the algorithm can be seen in [9].

Linear PCP Verifier algorithm $\mathcal{V}^{\pi}(\vec{v})$: The work of the verifier is split into two parts: the query algorithm $\mathcal{Q}$ and the decision algorithm $\mathcal{D}$.

- $\mathcal{Q}$: First of all, a random element $\tau \in \mathbb{F}$ is generated. Then the following queries $\vec{q_i} \in \mathbb{F}^{3+n+(m+1)=4+n+m}$ are computed:

  1. $\vec{q_1} = [Z_S(\tau), 0, 0] || \vec{A}(\tau) || [0, 0, \ldots, 0]$,
  2. $\vec{q_2} = [0, Z_S(\tau), 0] || \vec{B}(\tau) || [0, 0, \ldots, 0]$,
  3. $\vec{q_3} = [0, 0, Z_S(\tau)] || \vec{C}(\tau) || [0, 0, \ldots, 0]$,
  4. $\vec{q_4} = [0, 0, 0] || [0, 0, \ldots, 0, 0, \ldots, 0] || [1, \tau, \ldots, \tau^m]$,
  5. $\vec{q_5} = [0, 0, 0] || [1, \tau, \ldots, \tau^k, 0, \ldots, 0] || [0, 0, \ldots, 0]$.

  The state information is $\vec{u} := [1, \tau, \tau^2, \ldots, \tau^k, Z_S(\tau)]$. The query results are $a_i = \langle \vec{\pi}, \vec{q_i} \rangle$ for $i \in \{1, \ldots, 5\}$. Everything can be computed in $O(|\mathcal{C}|)$. Details of the algorithm can be seen in [9].

- $\mathcal{D}(\vec{v}, \vec{u}, \vec{a})$: Let $\vec{u} = [u_1, \ldots, u_{k+2}]$, $\vec{a} = [a_1, \ldots, a_5]$. The algorithm accepts iff the following two equalities hold:

  1. $a_1 a_2 - a_3 - a_4 u_{k+2} = 0$,
  2. $a_5 - u_1 - \langle \vec{v}, [u_2, \ldots, u_{k+1}] \rangle = 0$.

  These two checks verify that $P_1(\tau) = 0$ and $P_2(\tau) = 0$ for certain polynomials P_1 and P_2. The Schwarz-Zippel lemma ensures that proving $P(\tau)$ for a uniformly sampled τ is sufficient to prove that $P(x)$ is a zero polynomial. After all additional values related to Z_S, $\vec{h}$, $\delta_A, \delta_B, \delta_C$ are canceled out, we can see that $P_1(x) = 0$ (the first equality) proves $A\vec{w} \cdot B\vec{w} - C\vec{w} = \vec{0}$, and $P_2(x) = 0$ (the second equality) proves that the first k entries of $\vec{w}$ equal to the first k entries of $\vec{v}$.

  The randomness δ_A, δ_B, δ_C makes the proof a statistical HVZK proof. Namely, this randomness ensures that the scalar products $\langle \vec{\pi}, \vec{q}_i \rangle$ do not leak information about $\vec{w}$.

In Chapter 4, we will implement this particular LPCP using SMC. The verifier algorithm $\mathcal{V}^{\pi}(\cdot)$ will be implemented in a distributed way, so that no information about $\vec{v}$ is leaked to the adversary, while keeping the proof bound to the particular committed vector $\vec{v}$. The honest majority assumption ensures that the distributed $\mathcal{V}^{\pi}(\cdot)$ acts as an honest verifier, so that we may use the HVZK property.

CHAPTER 3

RELATED WORK

3.1 Actively and Covertly Secure Multiparty Computation

Several techniques exist for multiparty computation secure against active adversaries. There are implementations based on garbled circuits (GC) [58, 83], on additive sharing with message authentication codes (MACs) to check for correct behaviour [33, 31, 35], on Shamir's secret sharing [99, 28], and on the GMW protocol [47] paired with actively secure oblivious transfer (OT) [83]. Different techniques are suitable for different kinds of computations.

We give a brief overview of the GMW protocol in Section 3.1.2. More details about MACs and secret sharing can be found in Sections 2.3.3 and 2.4 respectively, and we give a particular protocol set using these techniques in Section 3.1.3. The details of GC and OT are not relevant for this thesis. The verification method that we propose in this thesis is mostly suitable for secret-sharing based SMC, with no preference towards the algebraic structures underlying the computation.

In this section, we describe in more detail some interesting sharing-based SMC protocols that share some common features with our work. We briefly explain which components we reuse, and what the main difference is from their usage in related work.

3.1.1 A Note on Covert Adversaries

Protocols that are secure against a covert adversary are in general much faster than fully actively secure protocols. The main techniques for achieving covert security are based on cut-and-choose, where the parties perform extra computation on dummy inputs, opening the dummy results afterwards to let the other parties check whether the computation was correct. These kinds of adversaries have been considered for example in [4, 29, 31, 70, 57]. In general, one does not aim to

achieve negligible cheating probability, and the covert adversary will not cheat even if the probability of detection equals a sufficiently large positive constant.

While in this thesis we are dealing with post-computation verification mechanisms, our ultimate goal is to achieve not *covert*, but *active* security. In all intermediate constructions achieving covert security, we assume that cheating is possible only with negligible probability. The work on covert adversaries in settings most similar to ours is [29], and we even borrow one functionality from them, but their probability of cheating is constant and cannot be made negligible without superpolynomial computational overhead.

3.1.2 Compilers from Passive to Active Security

One of the first SMC protocols was proposed by Goldreich et al. [47]. In this protocol, the function that the parties compute is defined by a boolean circuit, and special computation prescriptions are provided for the addition (exclusive OR) gates, and the multiplication (AND) gates. In its original form, the protocol is only secure against passive adversaries, since there is no way to check whether the parties have followed the protocol. A generic way to achieve active security is to use zero-knowledge proofs [45] to show that the protocol is being followed [46, Chapter 7.4]. These proofs are in general very complex, but they can be more efficient if we make some additional assumptions. We will use a similar approach in this thesis, appending a customized zero-knowledge verification phase to passively secure protocols.

More elaborate methods for post-execution verification of the correct behaviour of protocol participants have been presented in [29, 5]. We note that the general outline of our verification scheme is similar to [5]. We both commit to certain values during protocol execution and perform computations with them afterwards. However, the committed values and the underlying commitment scheme are very different. Honest majority assumption allows us to use more efficient linear threshold commitments (introduced in Section 2.6). Another important difference is that our solution can be straightforwardly applied to computation over rings.

3.1.3 Active Security for any Number of Corrupted Parties

The multiplication operation is often assisted by Beaver triples (Section 2.5). Such triples are used by several existing SMC frameworks, including ABY [38] or SPDZ [33]. We give a brief description the SPDZ protocol. It makes use of the following techniques.

- *Message authentication codes* (Section 2.3.3) allow to verify whether the shares have been affected by a malicious adversary.

- *Beaver triples* (Section 2.5) reduce multiplication to linear combinations.
- *Somewhat homomorphic encryption* is used in the preprocessing phase to compute the MAC key and Beaver triple shares. We only note that this technique involves expensive cryptographic operations, although it is much faster than *fully* homomorphic encryption, and we do not explain it in detail, as it is not essential in the context of our work.

SPDZ uses two kinds of additive sharings that allow message checking. The generation of these shares is described in more detail in [33]. There is a global MAC key α. It remains private throughout the computation, and it will be opened in the end to check whether the computation was correct.

1. The first sharing is

$$\langle a \rangle = (\delta, (a_1, \ldots, a_n), (\gamma(a)_1, \ldots, \gamma(a)_n)) \ ,$$

   where $a_1 + \cdots + a_n = a$ and $\gamma(a)_1 + \cdots + \gamma(a)_n = \alpha \cdot (a + \delta)$ for the global key α and a public value δ. The party P_i holds the pair $(a_i, \gamma(a)_i)$. Here $\gamma(a)_i$ are just additive shares of the value $\alpha \cdot (a + \delta)$, and their interpretation is that $\gamma(a) := \gamma(a)_1 + \cdots + \gamma(a)_n$ is the MAC authenticating the message a under the global key α.

   For two values a and b we have $\langle a \rangle + \langle b \rangle = \langle a + b \rangle$, where $\langle a + b \rangle$ is obtained by adding $\langle a \rangle$ and $\langle b \rangle$ componentwise, i.e. $\langle a + b \rangle = (\delta_a + \delta_b, (a_1 + b_1, \ldots, a_n + b_n), (\gamma(a)_1 + \gamma(b)_1, \ldots, \gamma(a)_n + \gamma(b)_n))$. For a constant c, we have $c + \langle a \rangle = (\delta - c, (a_1 + c, a_2, \ldots, a_n), (\gamma(a)_1, \ldots, \gamma(a)_n))$. Hence, any linear combination can be computed locally, directly on the shares.

   For this sharing, a *partial opening* is defined: each party P_i sends a_i to some fixed party (e.g. P_1) who computes $a = a_1 + \ldots + a_n$ and broadcasts a to all parties. The correctness of opening may be checked later by opening the MACs $\gamma(a)_i$, verifying that $\gamma(a)_1 + \cdots + \gamma(a)_n = \alpha \cdot (a + \delta)$.

2. The second sharing is

$$[\![a]\!] = ((a_1, \ldots, a_n), (b_1, \ldots, b_n), (\gamma(a)_1^i, \ldots, \gamma(a)_n^i)_{i \in [n]}) \ ,$$

   where $a = a_1 + \cdots + a_n$ and $\gamma(a)_1^i + \cdots + \gamma(a)_n^i = a \cdot b_i$. The party P_i holds the values $a_i, b_i, \gamma(a)_i^1, \ldots, \gamma(a)_i^n$. The idea is that $\gamma(a)_i := \gamma(a)_1^i + \cdots + \gamma(a)_n^i$ is the MAC authenticating a under the private key b_i of P_i. To open $[\![a]\!]$, each party P_j sends to each other party P_i its share a_j of a and its share $\gamma(a)_j^i$ of the MAC on a computed with the private key b_i of P_i. P_i checks that $\sum_{j \in [n]} \gamma(a)_j^i = a \cdot b_i$. To open the value to only one party P_i, the other parties simply send their shares only to P_i, who checks them. Only shares of a and $\gamma(a)_i = a \cdot b_i$ are needed for that.

In the specification of the protocol, it is assumed for simplicity that a broadcast channel is available, that each party has only one input, and only one public output value has to be computed. The number of input and output values can be generalized to an arbitrary number, without affecting the overall complexity, as shown in [33]. The protocol works as follows.

Initialization. Parties invoke preprocessing to get:

- the shared secret key $[\![\alpha]\!]$;
- a sufficient number of Beaver triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$;
- a sufficient number of pairs of random values $\langle r\rangle$, $[\![r]\!]$;
- a sufficient number of single random values $[\![t]\!]$, $[\![e]\!]$.

Generation of all these values is presented in more detail in [33]. It is based on a somewhat homomorphic encryption scheme. This makes the initialization phase quite expensive.

Inputs. If the party P_i provides an input x_i, a preshared pair $\langle r\rangle$, $[\![r]\!]$ is taken, and the following happens.

1. $[\![r]\!]$ is opened to P_i.
2. P_i broadcasts $x_i' = x_i - r$.
3. The parties compute $\langle x_i\rangle = \langle r\rangle + x_i'$.

Addition. In order to add $\langle x\rangle$ and $\langle y\rangle$, compute locally $\langle x + y\rangle = \langle x\rangle + \langle y\rangle$.

Multiplication. To multiply $\langle x\rangle$ and $\langle y\rangle$ the parties do the following.

1. Take two triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$ from the set of the available ones and check that indeed $a \cdot b = c$. This can be done as follows.
   - Open a random value $[\![t]\!]$, receiving t.
   - Partially open $a' = t \cdot \langle a\rangle - \langle f\rangle$ and $b' = \langle b\rangle - \langle g\rangle$.
   - Evaluate $t \cdot \langle c\rangle - \langle h\rangle - b' \cdot \langle f\rangle - a' \cdot \langle g\rangle - a' \cdot b'$, and partially open the result.
   - If the result is not zero the protocol aborts, otherwise go on with $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$.

   The idea is that, as t is random, it is difficult for the adversary to generate malicious shares such that the result is 0. This check can be done as a part of the preprocessing, for all triples in parallel, and, hence, only one random value t is sufficient.

2. Partially open $x' = \langle x \rangle - \langle a \rangle$ and $y' = \langle y \rangle - \langle b \rangle$. Compute $\langle z \rangle = x' \cdot y' + x' \cdot \langle b \rangle + y' \cdot \langle a \rangle + \langle c \rangle$.

Outputs. The output stage starts when parties already have $\langle y \rangle$ for the output value y, but this value has not been opened yet. Before the opening, it should be checked if all the parties have behaved honestly.

- Let $a_1, \ldots, a_T$ be all values publicly opened so far, where

$$\langle a_j \rangle = (\delta_j, (a_{j1}, \ldots, a_{jn}), (\gamma(a_j)_1, \ldots, \gamma(a_j)_n)) \ .$$

  The parties open a new random value $[\![e]\!]$, and set $e_i = e^i$ for all $i \in [T]$ (here e^i denotes the i-th power of e). All parties compute $a = \sum_{j \in [T]} e_j \cdot a_j$.
- Each P_i commits to $\gamma_i = \sum_{j \in [T]} e_j \cdot \gamma(a_j)_i$. For the output value $\langle y \rangle$, P_i also commits to the shares $(y_i, \gamma(y)_i)$ in the corresponding MAC of $\langle y \rangle$.
- $[\![\alpha]\!]$ is opened.
- Each P_i opens the commitment γ_i, and all parties check that $\alpha(a + \sum_{j \in [T]} e_j \cdot \delta_j) = \sum_{i \in [n]} \gamma_i$. If the check does not pass, the protocol aborts. Otherwise, the parties conclude that all the messages a_j are correct.
- To get the output value y, the commitments to $(y_i, \gamma(y)_i)$ are opened. Now y is defined as $y := \sum_{i \in [n]} y_i$, and each player checks that $\alpha(y + \delta) = \sum_{i \in [n]} \gamma(y)_i$. If the check passes, then y is the output.

This process verifies that all the intermediate values a_j, and also y, have indeed all been computed correctly.

While such methods can be secure for a dishonest majority, they lead to protocols that are in some sense weaker than ours. They do not allow the identification of a misbehaving party. In general, such protocols need to be extended with a possibility of an *identifiable abort* [49], making the honest parties blame a particular corrupted party that has caused the protocol to abort. Recently, some identification mechanisms for SPDZ-like protocols have been proposed [100, 27, 7], but the complexity of determining the identity of a misbehaving party may be too high for being a sufficient deterrent.

Another challenge is the difficulty of generating Beaver triples. The problem is that they must be private. Heavyweight cryptographic tools are used to generate them under the same privacy constraints as obeyed by the main phase of the protocol. Existing frameworks utilize homomorphic [38, 84, 91] or somewhat

(fully) homomorphic encryption systems [31, 20] or oblivious transfer [83]. Recently [54], the oblivious transfer methods of [40] have been extended to construct SPDZ multiplication triples over finite fields $\mathbb{Z}_p$, which made their generation significantly faster. However, these techniques only work for finite fields, not rings. To ensure the correctness of tuples, the generation is followed by a much cheaper correctness check [31].

In this thesis, we keep the correctness check, but the generation will be done in the open by the party whose behaviour is going to be checked. Differing from SPDZ, we use the triples, (and analogous tuples for other operations) not for performing computations, but for verifying them. This idea allows us to sidestep the most significant difficulties in precomputing the tuples. We note that a similar idea appeared in [41], where the triples are used to verify whether multiplications are computed correctly, but without pointing out the cheater. We discuss that work in the next subsection.

3.1.4 Active Security with an Honest Majority

For honest majority and three parties, a recent method [41], developed concurrently with our work, proposes a highly efficient actively secure protocol that is also based on precomputed multiplication triples. We will describe in more details how this protocol works.

The passively secure version of the protocol (without triple generation) was first proposed in [3]. It is based on additive sharing over finite rings $\mathbb{Z}_{2^m}$ among three *computing* parties. The number of parties providing inputs or receiving outputs may be much larger. Typically, the rings represent integers of certain length. The protocol set tolerates one passive corruption. It makes use of the following techniques.

- *Correlated randomness:* for every multiplication gate, the parties P_1, P_2, P_3 are given correlated randomness in the form of random ring elements x_1, x_2, x_3 such that $x_1 + x_2 + x_3 = 0$.
- *Linear threshold sharing:* each value v is shared in pairs (x_1, a_1), (x_2, a_2), (x_3, a_3), where $v = x_1 + x_2 + x_3 = x_1 - a_2 = x_2 - a_3 = x_3 - a_1$. In this way, any two parties are able to reconstruct v.

The protocol works as follows.

Initialization. Parties invoke the preprocessing phase to get random ring elements x_1, x_2, x_3 such that $x_1 + x_2 + x_3 = 0$ for each multiplication gate. The generation of such randomness is very cheap.

Inputs. Let v be the input that will be shared among P_i. It is done using a 2-out-of-3 secret sharing scheme as follows.

- P_1's share is the pair (x_1, a_1) where $a_1 = x_3 - v$;
- P_2's share is the pair (x_2, a_2) where $a_2 = x_1 - v$;
- P_3's share is the pair (x_3, a_3) where $a_3 = x_2 - v$.

Addition. Let $(x_1, a_1), (x_2, a_2), (x_3, a_3)$ be a secret sharing of v_1, and let (y_1, b_1), (y_2, b_2), (y_3, b_3) be a secret sharing of v_2. In order to compute a secret sharing of $v_1 + v_2$, each P_i locally computes (z_i, c_i) with $z_i = x_i + y_i$ and $c_i = a_i + b_i$.

Multiplication. Let (x_1, a_1), (x_2, a_2), (x_3, a_3) be a secret sharing of v_1, and let (y_1, b_1), (y_2, b_2), (y_3, b_3) be a secret sharing of v_2. It is assumed that the parties P_1, P_2, P_3 hold correlated randomness α, β, γ respectively, where $\alpha + \beta + \gamma = 0$. The parties compute the shares of $v_1 \cdot v_2$ as follows.

Step 1: Let 3^{-1} denote the inverse of 3 in $\mathbb{Z}_{2^m}$. The following messages are computed and sent in parallel:

- P_1 computes $r_1 = (x_1 \cdot y_1 - a_1 \cdot b_1 + \alpha) \cdot 3^{-1}$, and sends r_1 to P_2.
- P_2 computes $r_2 = (x_2 \cdot y_2 - a_2 \cdot b_2 + \beta) \cdot 3^{-1}$, and sends r_2 to P_3.
- P_3 computes $r_3 = (x_3 \cdot y_3 - a_3 \cdot b_3 + \gamma) \cdot 3^{-1}$, and sends r_3 to P_1.

Step 2: The following values are computed locally:

- P_1 stores (z_1, c_1), where $z_1 = r_3 - r_1$ and $c_1 = -2r_3 - r_1$;
- P_2 stores (z_2, c_2), where $z_2 = r_1 - r_2$ and $c_2 = -2r_1 - r_2$;
- P_3 stores (z_3, c_3), where $z_3 = r_2 - r_3$ and $c_3 = -2r_2 - r_3$.

Outputs. Each party P_i outputs z_i.

In consequent work [41], this passively secure protocol is extended with a Beaver triple based verification. After the computation, before any values are output, the correctness of each multiplication $x \cdot y = z$ is verified by a precomputed Beaver triple (r_x, r_y, r_z) similarly to the pairwise verification of SPDZ. Using threshold sharing ensures that the output cannot be modified by tampering with the share of the corrupted party, and it may at most lead to inconsistent opening, which results in the protocol aborting. The triples can be generated using the passively secure protocol described above, without relying on heavy cryptographic techniques like somewhat homomorphic encryption. The triples are then verified using cut-and-choose and pairwise verification, similarly to SPDZ.

Again, this method only allows the detection of misbehaviour, but no identification of the guilty party. Hence, this method is not applicable to a covert adversary, and the verification should take place at least after each declassification to ensure security against an active adversary. In our work, we will use Beaver triples to verify the multiplication gates corresponding to the *local* computation of each party. This allows us to generate the initial triples more efficiently since they should not remain private to the prover anymore, and he can generate the tuples himself instead of running a passively secure protocol. While the efficiency gain is not too high for common multiplication triples, delegating work to the prover becomes more important when generating more complex types of precomputed tuples.

3.1.5 Passive Security with an Honest Majority

The basic passively secure protocol of [3] is quite similar to Sharemind [16], which is also based on additive sharing over finite rings $\mathbb{Z}_{2^m}$ among three computing parties, tolerating one corrupted party. The basic Sharemind protocol works as follows.

Initialization. Parties invoke the preprocessing phase to get random ring elements x_1, x_2, x_3 such that $x_1 + x_2 + x_3 = 0$ for each multiplication gate. The generation of such randomness is very cheap.

Inputs. Let v be the input that will be shared among P_i. It is done using additive secret sharing as $v = x_1 + x_2 + x_3$, where x_i is given to P_i.

Addition. Let x_1, x_2, x_3 be a secret sharing of v_1, and let y_1, y_2, y_3 be a secret sharing of v_2. In order to compute a secret sharing of $v_1 + v_2$, each P_i locally computes $z_i = x_i + y_i$.

Multiplication. Let x_1, x_2, x_3 be a secret sharing of v_1, and let y_1, y_2, y_3 be a secret sharing of v_2. It is assumed that the parties P_1, P_2, P_3 hold correlated randomness $\alpha_1, \beta_1, \gamma_1$ respectively, where $\alpha_1+\beta_1+\gamma_1 = \alpha_2+\beta_2+\gamma_2 = 0$. The parties compute the shares of $v_1 \cdot v_2$ as follows.

Step 1: The following messages are computed and sent in parallel:

- P_1 computes $r_1 = x_1 + \alpha_1$ and $s_1 = y_1 + \alpha_2$. It sends r_1 to P_2, and s_1 to P_3.
- P_2 computes $r_2 = x_2 + \beta_1$ and $s_2 = y_2 + \beta_2$. It sends r_2 to P_3, and s_2 to P_1.
- P_3 computes $r_3 = x_3 + \gamma_1$ and $s_3 = y_3 + \gamma_2$. It sends r_3 to P_1, and s_3 to P_2.

Step 2: The following values are computed locally:

- P_1 stores $z_1 = r_1 \cdot s_1 + r_3 \cdot s_1 + r_3 \cdot s_2$;
- P_2 stores $z_2 = r_2 \cdot s_2 + r_1 \cdot s_2 + r_1 \cdot s_3$;
- P_3 stores $z_3 = r_3 \cdot s_3 + r_2 \cdot s_3 + r_2 \cdot s_1$.

Outputs. Each party P_i outputs z_i.

Although the online phase of the multiplication operation is more efficient for [16] if compared straightforwardly, Sharemind derives its efficiency from the great variety of protocols [17, 68, 59, 56] for integer, fix- and floating point operations, as well as for shuffling the arrays. The deployments of Sharemind [18, 51, 101] include the largest SMC applications ever [11, 12].

At the time of writing this thesis, only passive security was available for Sharemind. An auditability mechanism for Sharemind has been proposed and implemented in [87]. The idea of auditability is that, after the protocol execution has been finished, each party is assigned its own auditor who gets the local transcript of the party's computation and verifies that the computation has been correct. For this, the auditor needs to see all data of the party that he audits. In order to achieve active security with identifiable abort, the parties should be able to verify computation of each other by themselves, without help of external auditors, and without revealing their data to each other. In this thesis, we design a verification mechanism that we apply to the passively secure protocols of Sharemind, turning them to covertly and actively secure.

It would be interesting to apply our verification to [3] to enhance it with an identifiable abort that [41] does not provide. However, the protocol variety is currently much richer for Sharemind, and there are many more possible ways of optimizing specific protocols.

3.2 Multiple Adversary Models

There exist some alternative models, analogous to the UC model (described in Section 2.2.3), that support multiple adversaries, such as collusion preserving (CP) computation [1] or local UC (LUC) [23]. Both models are stronger than UC and allow to express more interesting properties.

Our initial idea was to take one of these models straightforwardly and apply it for our needs concerning detection of leakage from one honest party to another honest party. However, it turned out that both models are too strong, and some protocols that we intuitively treated as acceptable would be immediately ruled out by these models.

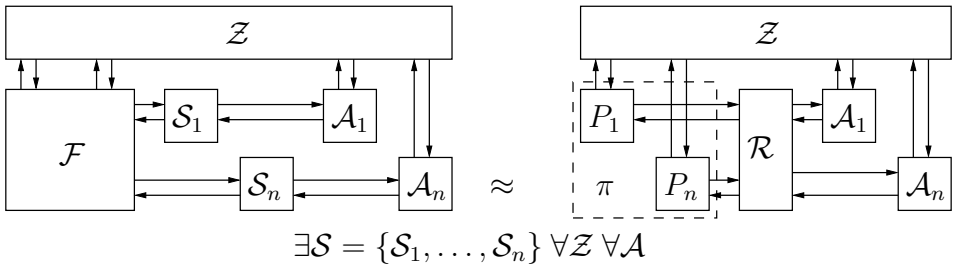


Figure 3.1: CP realization: the protocol π CP-realizes $\mathcal{F}$

3.2.1 Collusion Preserving Computation

We base our work on the collusion preserving (CP) computation of [1]. Although CP is based on *generalized universal composability* (GUC) [22], which assumes that the protocols may use some shared global setup, we give a simplified definition based on common UC.

Similarly to UC, there are ITMs $P_1, \ldots, P_n$ interacting with each other, the environment $\mathcal{Z}$, and the adversary $\mathcal{A}$. However, instead of one monolithic adversary $\mathcal{A}$, there are n adversaries $\mathcal{A}_1, \ldots, \mathcal{A}_n$, one for each party. It is assumed that $\mathcal{A}_i$ does not interact with P_i directly, but by means of some fixed communication resource R. All the adversaries are connected with the environment $\mathcal{Z}$, and, hence, potentially may use it for communicating with each other. The model is depicted in Figure 3.1.

In the definition of CP emulation, the simulator $\mathcal{S}$ should be of the form $\mathcal{S}_1, \ldots, \mathcal{S}_n$, where $\mathcal{S}_i$ mediates the communication between $\mathcal{A}_i$ and P_i or the communication resource R. It is important that different simulators $\mathcal{S}_i$ cannot communicate with each other, which makes them weaker compared to a monolithic simulator $\mathcal{S}$. Proving that a protocol π is as secure as a protocol ϕ now requires that the view of *each* party should be the same in π and ϕ. This immediately implies *collusion preservation*, i.e. any existing side-channel between any pair of adversaries $\mathcal{A}_i$ and $\mathcal{A}_j$ can be amplified by π no more than it could be amplified by ϕ. The security in the CP model is achievable, but it is quite complicated, as any subliminal channels that are not covered by ϕ should be eliminated from the protocol π to forbid additional communication between the corrupted parties (see [1] for details).

3.2.2 Local Universal Composability

Another model that supports multiple adversaries is Local UC (LUC) [23]. There is an adversary $\mathcal{A}_{(i,j)}$ for each ordered pair of parties P_i and P_j, residing on the communication channel from P_j to P_i. The model is depicted in Figure 3.1.

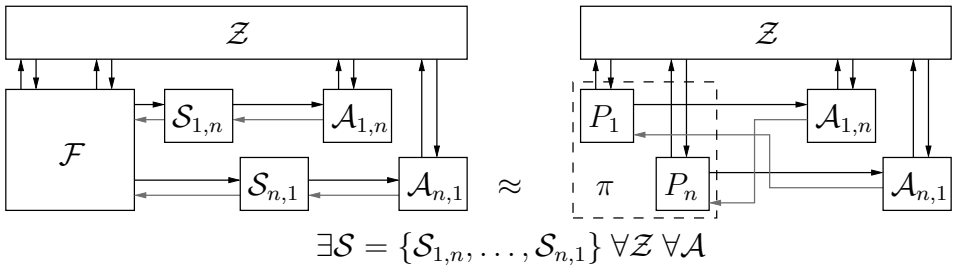


Figure 3.2: LUC realization: the protocol π LUC-realizes $\mathcal{F}$

Each party P_i may be corrupted by $n-1$ adversaries $\mathcal{A}_{(i,j)}$ that can deliver messages to the party P_i, where the sender identity of the delivered messages must be P_j. Proving that a protocol π is as secure as a protocol ϕ now requires that in π each entity affects each other entity in the same way as in ϕ. This model can be used to express more interesting properties than CP allows, such as anonymity, deniability, confinement (see [23] for more details).

3.3 Private Conditionals in SMC Programs

The support of conditionals with choices that depend on private data is present in SMCL [82], as well as in newer languages and frameworks, such as PICCO [106], Obliv-C [105], Wysteria [92], SCVM [73], or the DSL embedded in Haskell by Mitchell et al. [79]. A necessary precondition of making private conditionals possible is forbidding any public side effects inside the private branches (such as assignments to public variables or termination), since that may leak information about which branch has been executed. All the branches are executed simultaneously, and the value of each variable that could have been modified in at least one branch is updated by selecting its value obliviously, i.e. in such a way that no party knows which value has been chosen. Planul and Mitchell [88] have more thoroughly investigated the leakage through conditionals. They have formally defined the transformation for executing all branches and investigated the limits of its applicability to programs that have potentially non-terminating sub-programs.

The existing compilers that support private conditionals by executing all branches do not attempt to reduce the computational overhead of such an execution. We are aware of only a single optimization attempt targeted towards these sorts of inefficiencies [55]. They are targeting privacy-preserving applications running on top of garbled circuits, building a circuit into which all circuits representing the branches can be embedded. Their technique significantly depends on what can be hidden by the garbled circuit protocols about the details of the circuits. Our approach is more generic and applies at the language level.

CHAPTER 4

VERIFIABLE SMC WITH AN HONEST MAJORITY

4.1 Chapter Overview

In this chapter we propose a transformation for turning any passively secure multiparty protocol to a protocol that is covertly, or even actively secure under the honest majority assumption. The entire construction constitutes a variant of the GMW compiler (Section 3.1.2) from passively to actively secure protocols. Our verification phase can be seen as an interactive proof, where the verifier has been implemented using SMC to ensure its correct behaviour and prover's privacy. The main ideas behind our mechanism are the following:

- All the inputs and the incoming/outgoing messages of the prover are secret-shared among all the other parties using a threshold linear secret-sharing scheme. The verifiers repeat the prover's computations, using verifiable hints from the prover. The verification is zero-knowledge to any minority coalition of parties.
- The prover's hints are based on precomputed multiplication triples (Section 2.5), adapted for verification. Before starting the verification, and even the execution, the prover generates sufficiently many such triples and shares them among the other parties. Importantly, the prover provides a proof that these triples are generated and shared correctly. During verification, the correctness of triples implies the correctness of prover's computation.

Applying this verification mechanism n times to any n-party computation protocol, with each party acting as the prover in one instance, gives us a protocol secure against covert adversaries corrupting a minority of parties. Applying the verification after each round would result in an actively secure protocol.

4.2 The Ideal Functionality for Verifiable Honest Majority SMC

In this section, we formalize the initial passively secure protocol and specify the desired functionality of the resulting verifiable protocol in the UC framework (see Section 2.2.3). Such specification allows us to precisely state the security properties of the execution.

The initial passively secure protocol. The protocol is run by n parties, indexed by $[n]$, where $\mathcal{C} \subseteq [n]$ denotes the set of corrupted parties, $|\mathcal{C}| < n/2$. We denote $\mathcal{H} = [n] \backslash \mathcal{C}$. There is a secure channel between each pair of parties. The protocol is synchronous. It has r rounds, where the ℓ-th round computations of the party P_i, the results of which are sent to the party P_j, are given by a publicly known arithmetic circuit C_{ij}^{ℓ}. This circuit computes the ℓ-th round messages $\vec{m}_{ij}^{\ell}$ to the party $j \in [n]$ from the input $\vec{x}_i$, uniformly distributed randomness $\vec{r}_i$ and the messages $\vec{m}_{j'i}^{k}$ ($k < \ell$) that P_i has received before. All values $\vec{x}_i, \vec{r}_i, \vec{m}_{ij}^{\ell}$ are vectors over rings $\mathbb{Z}_{2^N}$. The messages received during the r-th round comprise the *output of the protocol*.

Arithmetic circuits C_{ij}^{ℓ} over rings $\mathbb{Z}_{2^{n_1}}, \ldots, \mathbb{Z}_{2^{n_K}}$ represent *local* computation of parties. Such a circuit consists of connected gates, performing arithmetic operations on inputs and producing outputs. An operation may be one of the following:

- an addition, a constant multiplication, or a multiplication in a ring $\mathbb{Z}_{2^{n_k}}$;
- the operation "$x = \mathsf{trunc}(y)$" for $x \in \mathbb{Z}_{2^n}$, $y \in \mathbb{Z}_{2^m}$, $n < m$, that computes $x = y \bmod 2^n$;
- the operation "$y = \mathsf{zext}(x)$" for $x \in \mathbb{Z}_{2^n}$, $y \in \mathbb{Z}_{2^m}$, $n < m$, that lifts $x \in \mathbb{Z}_{2^n}$ to the larger ring $\mathbb{Z}_{2^m}$;
- the operation $(z_1, \ldots, z_m) = \mathsf{bd}(x)$ that decomposes $x \in \mathbb{Z}_{2^m}$ into bits $z_i \in \mathbb{Z}_2$.

Gate outputs can be used as inputs of some other gates. The gate inputs that are not outputs of any other gates of C_{ij}^{ℓ} are called the *inputs* of C_{ij}^{ℓ}. Some gate outputs are treated as the final result of computing C_{ij}^{ℓ} on its inputs $\vec{x}$, and these are called the *outputs* of C_{ij}^{ℓ}. Computing the outputs $\vec{y}$ from the inputs $\vec{x}$ is denoted $\vec{y} = C_{ij}^{\ell}(\vec{x})$.

This set of gates is sufficient to represent any computation. Any other operations can be expressed as a composition of the available ones. Nevertheless, the verifications designed for special gates may be more efficient, and we discuss some of them in Section 4.6.1.

The resulting verifiable protocol. The verifiable protocol execution is specified by the ideal functionality $\mathcal{F}_{vmpc}$ given in Figure 4.1. Parties are given a set of publicly known arithmetic circuits C_{ij}^{ℓ} specifying the initial passively secure protocol. Honest parties use C_{ij}^{ℓ} to compute their outgoing messages m_{ij}^{ℓ}. The outgoing messages $m_{ij}^{*\ell}$ of corrupted parties are chosen by the adversary.

After the computation ends, $\mathcal{F}_{vmpc}$ outputs to *all* honest parties a set $\mathcal{M}$ containing *all* corrupted parties P_i that have sent $\vec{m}_{ij}^{*\ell} \neq \vec{m}_{ij}^{\ell}$ to any honest party P_j during the execution, and also *all* parties that have caused the protocol to abort (the set $\mathcal{B}_0$). Even if only *some* rounds of the protocol are computed, all the parties that deviated from the protocol in completed rounds will be detected.

The sets $\mathcal{B}_i$ of parties that are finally blamed by P_i may contain some additional parties that do not belong to $\mathcal{M}$. This is related to unsuccessful cheating that may have been detected only by some parties. Since $\mathcal{B}_i \subseteq \mathcal{C}$, no honest parties (in $\mathcal{H}$) can be falsely blamed.

We note that if $\mathcal{M} = \emptyset$, then $\mathcal{A}_S$ does not learn anything that a semi-honest adversary could not learn. In this way, the transformed protocol defines a covertly secure execution of the protocol specified by C_{ij}^{ℓ}: even though the corrupted parties may cheat, they will be finally detected if they do it.

Differently from the initial passively secure protocol, the parties are no longer trusted to generate their randomness $\vec{r}_i$ themselves. Instead, $\vec{r}_i$ is generated by $\mathcal{F}_{vmpc}$, before the parties get their inputs $\vec{x}_i$ from $\mathcal{Z}$. At this point, the adversary may stop the functionality. This corresponds to the failure of randomness generation in the real protocol, and it is allowed by $\mathcal{F}_{vmpc}$, since it is safe to abort the computation that does not involve private inputs.

4.3 The Protocol for Verifiable 3-Party SMC with one Corrupted Party

The initial passively secure protocol is defined by circuits C_{ij}^{ℓ} representing local computation of parties, as defined in Section 4.2. In order to get an implementation of $\mathcal{F}_{vmpc}$, we transform such a protocol to a verifiable one, outlined as follows.

At the beginning of the **execution phase**, P_i commits itself to its inputs $\vec{x}_i$ and the randomness $\vec{r}_i$. The commitment method ensures that $\vec{r}_i$ is distributed uniformly. Then the parties start executing the protocol defined by C_{ij}^{ℓ}. During the execution, P_i computes the messages $\vec{m}_{ij}^{\ell}$ using C_{ij}^{ℓ}, committing itself and the receiver P_j to them. If P_i and P_j are both corrupted, then they are allowed to commit to arbitrary $\vec{m}_{ij}^{*\ell}$. For $\mathcal{F}_{vmpc}$ it is sufficient that P_i and P_j commit to $\vec{m}_{ij}^{\ell}$ already after the execution phase ends, but if exactly one of them is honest, it should be able to prove which $\vec{m}_{ij}^{\ell}$ has actually been transmitted.

• **In the beginning**, $\mathcal{F}_{vmpc}$ gets from $\mathcal{Z}$ for each party P_i the message $(\mathsf{circuits}, i, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ and forwards it to $\mathcal{A}_S$.
For each $i \in [n]$, $\mathcal{F}_{vmpc}$ generates the randomness $\vec{r}_i$ for the party P_i. For $i \in \mathcal{C}$, it sends $(\mathsf{randomness}, i, \vec{r}_i)$ to $\mathcal{A}_S$.
At this point, $\mathcal{A}_S$ **may stop** the functionality. If it continues, then for each $i \in \mathcal{H}$ [resp $i \in \mathcal{C}$], $\mathcal{F}_{vmpc}$ gets the inputs $(\mathsf{input}, \vec{x}_i)$ for the party P_i from $\mathcal{Z}$ [resp. $\mathcal{A}_S$].

• **For each round** $\ell \in [r]$, $i \in \mathcal{H}$ and $j \in [n]$, $\mathcal{F}_{vmpc}$ uses C_{ij}^{ℓ} to compute the message $\vec{m}_{ij}^{\ell}$ that the party P_i is supposed to deliver to P_j on the ℓ-th round. For all $j \in \mathcal{C}$, it sends $\vec{m}_{ij}^{\ell}$ to $\mathcal{A}_S$. For each $j \in \mathcal{C}$ and $i \in \mathcal{H}$, it receives $\vec{m}_{ji}^{\ell}$ from $\mathcal{A}_S$.

• **After** r **rounds**, $\mathcal{F}_{vmpc}$ sends $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$ to each party P_i with $i \in \mathcal{H}$. Let $r' = r$ and $\mathcal{B}_0 = \emptyset$.
Alternatively, **at any time** before outputs are delivered to parties, $\mathcal{A}_S$ may send $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$, where $\mathcal{B}_0 \subseteq \mathcal{C}$ are the parties that caused the abort. In this case the outputs are not sent. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.

• **After** r' **rounds**, $\mathcal{A}_S$ sends to $\mathcal{F}_{vmpc}$ the messages $\vec{m}_{ij}^{\ell}$ for $\ell \in [r']$ and $i, j \in \mathcal{C}$.
$\mathcal{F}_{vmpc}$ defines the set of cheaters $\mathcal{M} = \mathcal{B}_0 \cup \{i \in \mathcal{C} \mid \exists j \in [n], \ell \in [r'] : \vec{m}_{ij}^{\ell} \neq C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})\}$.

• **Finally**, for each $i \in \mathcal{H}$, $\mathcal{A}_S$ sends $(\mathsf{blame}, i, \mathcal{B}_i)$ to $\mathcal{F}_{vmpc}$, with $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$. $\mathcal{F}_{vmpc}$ forwards this message to P_i.

Figure 4.1: The ideal functionality $\mathcal{F}_{vmpc}$ for verifiable computations

After the execution phase ends, the **verification phase** starts. Each party (the prover P) has to prove to the other parties (the verifiers $V_1, \ldots, V_{n-1}$) that it computed its local circuits C_{ij}^{ℓ} correctly w.r.t. committed $\vec{x}_i$, $\vec{r}_i$, $\vec{m}_{ij}^{\ell}$, $\vec{m}_{ki}^{\ell}$ for $k \in [n]$. All n interactive proofs of the n provers take place in parallel. It is possible that some verifiers V_i misbehave during the proof, and they should be blamed for that, even if they have not cheated during the execution phase. The proofs of all parties should terminate even if corrupted verifiers leave the protocol.

The previous discussion is summarized by the ideal functionality $\mathcal{F}_{verify}$ depicted in Figure 4.2. It treats all circuits C_{ij}^{ℓ} of one party P_i as a single circuit C_i. It assigns a unique index id to each input and output of the circuit. Such indexation makes it easier to see which commitments correspond to which inputs and outputs of the circuit C_i.

In the rest of this section, we describe the protocol UC-realizing $\mathcal{F}_{verify}$, and the building blocks used by it. The protocol is going to have its own **preprocessing phase**, aiming to make the verification phase cheaper.

Throughout this section, we assume that the number of parties is 3, and at most one of them is corrupted. This assumption makes the presentation simpler, and it describes precisely our actual implementation, including all optimizations specific to 3 parties. We discuss in Section 4.4 how the transformation can be generalized to any number n of parties, still assuming an honest majority. In Section 4.5, we give formal definitions of n-party protocols and the corresponding proofs.

$\mathcal{F}_{verify}$ uses arrays $comm$ and $sent$ for storing the commitments. It works with unique indices id, defining a commitment $comm[id]$ and its ring size $m(id)$. The messages are first stored as $sent[id]$ before they are finally committed.

• **Initialization:** On input $(\mathsf{init}, (C_{ij}^{\ell})_{i,j,\ell}^{n,n,r})$ from all (honest) parties, where C_{ij}^{ℓ} is an arithmetic circuit, initialize $comm$ and $sent$ to empty arrays. For all $i \in [n]$, treat the composition of C_{ij}^{ℓ} for $j \in [n]$, $\ell \in [r]$ as a single circuit C_i. Generate a unique index xid_k^i for the k-th input of C_i, and yid_k^i for the k-th output of C_i. For all obtained indices id, read out from C_i the ring size $m(id)$ of the value indexed by id. Store C_i and all id, $m(id)$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, xid_k^i)$ from all (honest) parties, if $comm[xid_k^i]$ is not defined yet, generate $r \xleftarrow{\$} \mathbb{Z}_{m(xid_k^i)}$, and assign $comm[xid_k^i] \leftarrow r$. Output r to P_i. If $i \in \mathcal{C}$, output r also to $\mathcal{A}_S$.

• **Input Commitment:** On input $(\mathsf{commit\_input}, x, xid_k^i)$ from P_i, and $(\mathsf{commit\_input}, xid_k^i)$ from all (honest) parties, if $comm[xid_k^i]$ is not defined yet, assign $comm[xid_k^i] \leftarrow x$. If $i \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$.

• **Message Commitment:**

1. On input $(\mathsf{send\_msg}, x, yid_l^i, xid_k^j)$ from P_i, output x to P_j. If $i \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$. If $j \in \mathcal{C}$, output x to $\mathcal{A}_S$. If $sent[yid_l^i]$ is not defined yet, assign $sent[yid_l^i] \leftarrow x$.
2. On input $(\mathsf{commit\_msg}, yid_l^i, xid_k^j)$ from all (honest) parties, if $sent[yid_l^i]$ is defined, and $comm[yid_l^i]$ is not defined, assign $comm[yid_l^i] = comm[xid_k^j] \leftarrow sent[yid_l^i]$. If $i, j \in \mathcal{C}$, assign $comm[yid_l^i] = comm[xid_k^j] \leftarrow x^*$, where x^* is chosen by $\mathcal{A}_S$.

• **Verification:** On input (verify, i) from all (honest) parties, if $comm[id]$ has been defined for all identifiers id of C_i, construct vectors $\vec{x}$ and $\vec{y}$ such that $x_j \leftarrow comm[xid_j^i]$, and $y_j \leftarrow comm[yid_j^i]$. Compute $\vec{y'} \leftarrow C_i(\vec{x})$.
If $\vec{y'} - \vec{y} = \vec{0}$, output 1 to each party and $\mathcal{A}_S$. Otherwise, output 0 to each party and $\mathcal{A}_S$.

• **Cheater detection:** On input $(\mathsf{cheater}, k)$ from $\mathcal{A}_S$ for $k \in \mathcal{C}$, output $(\mathsf{cheater}, k)$ to each party. Do not accept any inputs from P_k anymore.

Figure 4.2: The ideal functionality $\mathcal{F}_{verify}$ for verifying circuit computation

4.3.1 Building Blocks

Ensuring Message Delivery. At any time during the protocol execution, a corrupted sender may refuse to send the message. If the receiver complains about not receiving it, the other parties do not know whether they should blame the sender or the receiver. It would be especially sad to allow a corrupted party to abort the verification phase in this way, so that the cheaters would not be pinpointed.

We want to achieve *identifiable abort*, i.e. if some party stops the protocol, it is blamed by all (honest) parties. For this purpose, we use the transmission functionality $\mathcal{F}_{transmit}$ proposed in [29] that we repeat in Figure 4.3. It allows to ensure message delivery, and to reveal previously received messages. The adversary may still interrupt transmission, but in this case a message $(\mathsf{cheater}, k)$ will be output to all honest parties, where $k \in \mathcal{C}$ has caused the interruption.

The protocol $\Pi_{transmit}$ implementing $\mathcal{F}_{transmit}$, also taken from [29], is given in Figure 4.4. All the messages have signatures so that they could be revealed later.

$\mathcal{F}_{transmit}$ works with unique message identifiers id, encoding a sender $s(id) \in [n]$ and a receiver $r(id) \in [n]$.

• **Initialization:** On input (init, s, r) from all (honest) parties, where s, r map a message identifier id to its sender and receiver respectively, deliver (init, s, r) to $\mathcal{A}_S$.

• **Secure transmit:** On input $(\mathsf{transmit}, id, m)$ from $P_{s(id)}$ and $(\mathsf{transmit}, id)$ from all (honest) parties, output (id, m) to $P_{r(id)}$, and $(id, |m|)$ to $\mathcal{A}_S$. If $r(id) \in \mathcal{C}$, output (id, m) to $\mathcal{A}_S$.

• **Reveal received message:** On input (reveal, id, i) from all (honest) parties, such that $P_{r(id)}$ at any point received (id, m), output (id, m) to P_i. If $i \in \mathcal{C}$, output (id, m) also to $\mathcal{A}_S$. If both $s(id), r(id) \in \mathcal{C}$, then $\mathcal{A}_S$ can ask $\mathcal{F}_{transmit}$ to output (id, m') for any m'.

• **Cheater detection:** If $\{s(id), r(id)\} \cap \mathcal{C} \neq \emptyset$, $\mathcal{A}_S$ may interrupt the transmission and ask $\mathcal{F}_{transmit}$ to output $(\mathsf{cheater}, k)$ to all parties for $k \in \mathcal{C} \cap \{s(id), r(id)\}$. If $(\mathsf{cheater}, k)$ is output for all $k \in \{s(id), r(id)\}$, then no (id, m) is output to the parties.

Figure 4.3: Ideal functionality $\mathcal{F}_{transmit}$

• **Initialization:** On input (init, s, r), the parties generate and exchange their public keys that will be used to verify signatures later.

• **Secure transmit:** *Cheap mode:* use as far as $P_{r(id)}$ does not complain.

1. On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s. It sends (id, m, σ_s) to $P_{r(id)}$.
2. On input $(\mathsf{transmit}, id)$ the party $P_{r(id)}$ expects a message (id, m, σ_s) from $P_{s(id)}$, where σ_s is a valid signature from $P_{s(id)}$ on (id, m). If it receives it, it outputs (id, m) to $\mathcal{Z}$. If it does not receive it within one round, it sends to each other party a signature $\gamma_{r(id)}$ on message $(\mathsf{bad}, s(id))$. Any party receiving $\gamma_{r(id)}$ sends it to all other parties.

Expensive mode: an honest party goes to expensive mode if it receives a signature $\gamma_{r(id)}$ on $(\mathsf{bad}, s(id))$ from at least t parties.

1. On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s. It sends (id, m, σ_s) to each other party.
2. Each party P_i sends (id, m, σ_s) to $P_{r(id)}$. If P_i does not receive (id, m, σ_s) within one round, it sends to $P_{r(id)}$ a signature γ_i on $(\mathsf{cheater}, s(id))$ instead.
3. On input $(\mathsf{transmit}, id)$, $P_{r(id)}$ expects a message (id, m, σ_s) from each P_i, where σ_s is a valid signature of $P_{s(id)}$ on (id, m). If it arrives from some P_i, then $P_{r(id)}$ outputs (id, m). Otherwise, $P_{r(id)}$ sends all γ_i that it has received to the other parties. The parties exchange γ_i, and any party receiving γ_i from at least t parties outputs $(\mathsf{cheater}, s(id))$ to $\mathcal{Z}$.

• **Reveal received message:** On input (reveal, id, i), the party $P_{r(id)}$ which at any point should have received the message (id, m, σ_s), sends $(\mathsf{reveal}, id, m, \sigma_s)$ to P_i. This is done analogously to secure transmission, and the message is treated as invalid if σ_m is not a valid signature of m. If $P_{r(id)}$ is detected in cheating, then $P_{s(id)}$ is allowed to reveal any m to P_i.

Figure 4.4: Real protocol $\Pi_{transmit}$

If some transmission aborts for unknown reasons, then the sender is required to deliver the message to *each other party*, so that at least one other honest party forwards it to the receiver. If no honest party receives the message, the sender will be blamed by all of them. This approach does not break data confidentiality in a single adversary model (like UC). The reason is that a message is published only if a conflict takes place between the sender and the receiver. In this case, at least one of them is corrupted, hence the adversary has seen that message anyway.

We use $\mathcal{F}_{transmit}$ not only in the execution, but also in the preprocessing and the verification phases, in order to ensure that all shares are delivered and the verification terminates. For simplicity, in this section we write that a message has been transmitted or revealed using $\mathcal{F}_{transmit}$, and avoid using its formal interface, since handling message identifiers requires introducing technical details.

Broadcast and opening. Broadcast with identifiable abort can be built on top of $\mathcal{F}_{transmit}$. If the party P wants to broadcast a message m, it uses $\mathcal{F}_{transmit}$ to deliver m to each other party. Upon receiving m_i and m_j respectively, the parties P_i and P_j exchange $h_i = H(m_i)$ and $h_j = H(m_j)$, where H is a collision-resistant hash function. If $h_i \neq h_j$, then both m_i and m_j are revealed to each party through $\mathcal{F}_{transmit}$, allowing to identify the cheater: note that if at least one of P, P_i, P_j is honest, it is impossible to reveal different messages to different parties. Since H is collision-resistant, $h_i = h_j$ implies $m_i = m_j$ with high probability, even if the adversary chooses m_i and m_j. Since hashing is only used for compactness, and h_i can be computed directly from m_i that is not private anyway, all hashes can be easily simulated in the security proofs.

Using the same idea, a previously transmitted message can be revealed to *all* parties by first using $\mathcal{F}_{transmit}$ to reveal the message to each party separately, and then exchange the hashes to ensure that each party has got the same value. Exchange of hashes is not necessary in the 3-party case, since at most one party is corrupted, and two different messages (id, m) and (id, m') cannot be revealed.

Throughout this section, by *broadcast* and by *opening* we mean these $\mathcal{F}_{transmit}$-based protocols. In order to avoid ambiguity, no other definitions of broadcast and opening are used.

Sharing Based Commitments. All the inputs, the randomness, and the messages of the prover P are committed by additively sharing them among the verifiers V_1 and V_2. To commit to $x \in \mathbb{Z}_m$, the prover P generates random $x^1 \xleftarrow{\$} \mathbb{Z}_m$ and computes $x^2 = x - x^1$ in $\mathbb{Z}_m$. Then P uses $\mathcal{F}_{transmit}$ to deliver x^i to V_i. Using $\mathcal{F}_{transmit}$ allows to argue about the authenticity of x^i later, if there are conflicts between P and V_i. We write $[\![x]\!]$ to denote the sharing of x, and $x = x^1 + x^2$ to denote that x was shared to the particular shares x^1 and x^2.

One instance of $\mathcal{F}_{pre}$ is used to generate u preprocessed tuples in a ring $\mathbb{Z}_m$ for one party P_i. It works with unique indices id defining a multiplication triple $\mathsf{triple}[id]$ or a trusted bit $\mathsf{bit}[id]$.

• **Initialization:** On input (init, i, m, u) from all (honest) parties, where $i \in [n]$ is a party index, initialize triple and bit to empty arrays. Generate u unique indices id. Store m, u, i and all id for further use. As shorthand notation, let $P = P_i$. Let V_1 and V_2 denote the other two parties.

• **Trusted bit generation:** On input (bit, id) from all (honest) parties, check if $\mathsf{bit}[id]$ exists. If it does, take $(\vec{b}^1, \vec{b}^2) \leftarrow \mathsf{bit}[id]$. Otherwise, generate a vector of random bits $\vec{b} \xleftarrow{\$} \mathbb{Z}_2^u$. If $i \in \mathcal{C}$, then $\vec{b} \in \mathbb{Z}_2^u$ is chosen by $\mathcal{A}_S$. Share elementwise $\vec{b} = \vec{b}^1 + \vec{b}^2$ over $\mathbb{Z}_m$. Assign $\mathsf{bit}[id] \leftarrow (\vec{b}^1, \vec{b}^2)$. Output $\vec{b}^j$ to V_j. Output $(\vec{b}^1, \vec{b}^2)$ to P. For $k \in \mathcal{C}$, send $\vec{b}^k$ to $\mathcal{A}_S$. If $i \in \mathcal{C}$, send $(\vec{b}^1, \vec{b}^2)$ to $\mathcal{A}_S$.

• **Multiplication triple generation:** On input (triple, id) from all (honest) parties, check if $\mathsf{triple}[id]$ exists. If it does, take $((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2)) \leftarrow \mathsf{triple}[id]$. Otherwise, generate random vectors $\vec{a} \xleftarrow{\$} \mathbb{Z}_m^u$, $\vec{b} \xleftarrow{\$} \mathbb{Z}_m^u$, and compute elementwise $\vec{c} \leftarrow \vec{a} \cdot \vec{b}$. If $i \in \mathcal{C}$, then $\vec{a}, \vec{b} \in \mathbb{Z}_m^u$ are chosen by $\mathcal{A}_S$. Share elementwise $\vec{a} = \vec{a}^1 + \vec{a}^2$, $\vec{b} = \vec{b}^1 + \vec{b}^2$, and $\vec{c} = \vec{c}^1 + \vec{c}^2$ over $\mathbb{Z}_m$. Assign $\mathsf{triple}[id] \leftarrow ((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2))$.

Output $(\vec{a}^j, \vec{b}^j, \vec{c}^j)$ to V_j. Output $((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2))$ to P. For $k \in \mathcal{C}$, send $(\vec{a}^k, \vec{b}^k, \vec{c}^k)$ also to $\mathcal{A}_S$. If $i \in \mathcal{C}$, send $((\vec{a}^1, \vec{b}^1, \vec{c}^1), (\vec{a}^2, \vec{b}^2, \vec{c}^2))$ to $\mathcal{A}_S$.

• **Stopping:** At any time, on input (stop) from $\mathcal{A}_S$, stop the functionality and output $\perp$ to all parties.

Figure 4.5: Ideal functionality $\mathcal{F}_{pre}$

Throughout this section, by *commitment* we mean this sharing-based commitment. In order to avoid ambiguity, no other definition of commitment is used.

Precomputed tuples. To reduce the work of the verifiers, we add a preprocessing phase generating correlated randomness, i.e. *precomputed tuples* (see Section 2.5). They are secret-shared among the verifiers, who have been convinced that the correlation holds. The prover P gets all the shares. The *verified multiplication triples* are triples (a, b, c) from some ring, such that $a \cdot b = c$. The *trusted bits* are values b from some ring $\mathbb{Z}_m$, $m > 2$, such that $b \in \{0, 1\}$. The preprocessing phase may fail, and it is possible that the deviator cannot be identified. This is not a problem since no private data is involved into this phase. We formalize this phase as a functionality $\mathcal{F}_{pre}$ given in Figure 4.5. We give the implementation of $\mathcal{F}_{pre}$ in Section 4.3.2.

4.3.2 Protocol Implementing $\mathcal{F}_{pre}$

The protocol Π_{pre} implementing $\mathcal{F}_{pre}$ is given in Figure 4.6. The prover P, allowed to know the sharings, generates and shares the bits and the triples itself. The shares are delivered to the verifiers through $\mathcal{F}_{transmit}$, so that P gets committed to the values it has generated. The prover is interested in generating the tuples randomly, because his (and only his) privacy depends on it. Since the prover generates the tuples itself, a corrupted verifier cannot provide invalid tuples for an honest prover.

• **Initialization:** The protocol starts with each party getting the input (init, i, m, u), where P_i is the prover, m is the ring size, and u is the number of tuples to be generated. The protocol uses parameters μ and κ that depend on the security parameter. As shorthand notation, let $P = P_i$. Let V_1 and V_2 denote the other two parties.

• **Trusted bits:** On input (bit, id):

1. The party P generates $(\mu \cdot u + \kappa)$ random bits $b \xleftarrow{\$} \mathbb{Z}_2$.
2. P shares $b = b^1 + b^2$ in $\mathbb{Z}_m$, and sends b^i to V_i using $\mathcal{F}_{transmit}$.
3. The parties agree on a public random permutation π of generated bits $\vec{b}$. For $b \in \{b_{\pi(1)}, \ldots, b_{\pi(\kappa)}\}$, V_1 and V_2 open b^1 and b^2 through $\mathcal{F}_{transmit}$, and each party computes $b = b^1 + b^2$. If $b \notin \{0, 1\}$, each party outputs $\perp$.
4. The remaining bits are split into groups of μ, where the first $\mu - 1$ bits are used to verify the μ-th one. Let the bit $[\![b']\!]$ be used to verify that $[\![b]\!]$ is a bit. P broadcasts a bit c indicating whether $b = b'$ or not. If $c = 1$ (indicating $b = b'$), the verifiers compute $[\![z]\!] = [\![b]\!] - [\![b']\!]$. If $c = 1$, the verifiers compute $[\![z]\!] = [\![b]\!] + [\![b']\!] - 1$.
5. After V_1 and V_2 have computed $[\![z]\!]$ for all bit pairs, they are holding the vector shares $\vec{z}^1$ and $\vec{z}^2$ respectively. They compute and exchange hashes $h_1 = H(\vec{z}^1)$ and $h_2 = H(-\vec{z}^2)$, checking if $h_1 = h_2$. If the check fails, V_1 and V_2 inform P about the failure, and each party outputs $\perp$. If it succeeds, V_1 and V_2 inform P about success. For each of the remaining u bits b, P outputs b, and V_1 and V_2 output the shares b^1 and b^2 respectively.

• **Multiplication triples:** On input (triple, id):

1. The party P generates $(\mu \cdot u + \kappa)$ triples (a, b, c) such that $a \xleftarrow{\$} \mathbb{Z}_m$, $b \xleftarrow{\$} \mathbb{Z}_m$, $c = a \cdot b$.
2. P shares $a = a^1 + a^2$, $b = b^1 + b^2$, $c = c^1 + c^2$ in $\mathbb{Z}_m$, and sends (a^i, b^i, c^i) to V_i using $\mathcal{F}_{transmit}$.
3. The parties agree on a public random permutation π of generated triples. For $a \in \{a_{\pi(1)}, \ldots, a_{\pi(\kappa)}\}$, $b \in \{b_{\pi(1)}, \ldots, b_{\pi(\kappa)}\}$, $c \in \{c_{\pi(1)}, \ldots, c_{\pi(\kappa)}\}$, V_1 and V_2 open a^1, b^1, c^1, a^2, b^2, c^2 through $\mathcal{F}_{transmit}$. Each party computes $a = a^1 + a^2$, $b = b^1 + b^2$, $c = c^1 + c^2$. If $a \cdot b \neq c$, each party outputs $\perp$.
4. The remaining triples are split into groups of μ, where the first $\mu - 1$ triples are used to verify the μ-th one. Let the triple $([\![a']\!], [\![b']\!], [\![c']\!])$ be used to verify the correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$. The verifiers compute $[\![\hat{a}]\!] = [\![a]\!] - [\![a']\!]$ and $[\![\hat{b}]\!] = [\![b]\!] - [\![b']\!]$, and declassify $\hat{a}, \hat{b}$ by exchanging the shares $\hat{a}^i = a^i - a'^i$ and $\hat{b}^i = b^i - b'^i$. Then they compute $[\![z]\!] = \hat{a} \cdot [\![b]\!] + \hat{b} \cdot [\![a']\!] + [\![c']\!] - [\![c]\!]$.
5. The checks $z = 0$ are done similarly to the step (5) of trusted bits. If the check fails, each party outputs $\perp$. If it succeeds, for each of the remaining u triples (a, b, c), P outputs (a, b, c), and V_1 and V_2 output the shares (a^1, b^1, c^1) and (a^2, b^2, c^2) respectively.

• **Stopping:** If at any time $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{transmit}$, each party outputs $\perp$.

Figure 4.6: Real protocol Π_{pre}

The verifiers check whether P generated the tuples correctly. The check is based on cut-and-choose and pairwise check, similarly to e.g. [33, 41]. The check is probabilistic, and it depends on parameters μ and κ. In order to obtain u tuples of certain kind, $\mu \cdot u + \kappa$ tuples have to be generated and shared by P.

First, the parties agree on a joint random seed, defining a random permutation π of the tuples. In the cut-and-choose step, they take the first κ randomly permuted tuples and open them. The check fails if any of the opened tuples is not correct. If all of them are correct, then only a negligible fraction of remaining tuples is wrong. This is not enough since we want *all* the tuples to be correct with high probability.

The remaining tuples are partitioned into groups of size μ. In each group, the first $\mu - 1$ tuples are used to verify the μ-th one in $\mu - 1$ pairwise checks. The core of each check is using homomorphic properties of secret sharing to compute a certain linear combination z of the tuple elements and verify that $z = 0$ (we call such z an *alleged zero*). The check is certain to fail if only one of the tuples in the pair is correct. Let $[\![z]\!]$ be computed as in Figure 4.6. We show that, if $z = 0$, and one tuple is correct, then the other tuple is certainly also correct.

Trusted bits. Let the bit $[\![b']\!]$ in a ring $\mathbb{Z}_m$ be used to verify that $[\![b]\!]$ is a bit. Let $b' \in \{0, 1\}$. The prover broadcasts a bit c indicating whether $b = b'$.

- If $c = 1$, then $[\![z]\!] = [\![b]\!] - [\![b']\!]$ is computed. If $z = 0$, then it should be $b = b' \in \{0, 1\}$.
- If $c = 0$, then $[\![z]\!] = [\![b]\!] + [\![b']\!] - 1$ is computed. If $z = 0$, then it should be $b = 1 - b' \in \{0, 1\}$.
- If $c \notin \{0, 1\}$, the protocol aborts.

Multiplication triples. Let the triple $([\![a']\!], [\![b']\!], [\![c']\!])$ be used to verify the correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$. Let $c' = a' \cdot b'$. The values $\hat{a} = a - a', \hat{b} = b - b'$ are computed and declassified by the verifiers, so there is no way for P to cheat with them. The verifiers compute and declassify $[\![z]\!] = \hat{a} \cdot [\![b]\!] + \hat{b} \cdot [\![a']\!] + [\![c']\!] - [\![c]\!]$. Since $c' = a' \cdot b'$, we have $z = \hat{a} \cdot b + \hat{b} \cdot a' + a' \cdot b' - c = a \cdot b - c$. Therefore, if $z = 0$, then $a \cdot b = c$. Opening the shares of z leaks no information of a honest prover, since one share z^i that belongs to the corrupted verifier V_i is already known to the adversary, and the other one could be computed as $z^j = -z^i$.

The bit c denoting whether $b = b'$ and the values $\hat{a} = a - a', \hat{b} = b - b'$ are all distributed uniformly in the corresponding rings, since one of the tuples serves as a mask for the other tuple.

In the protocol of Figure 4.6, the verifiers do not check $\vec{z} = \vec{0}$ directly. Instead, they exchange $h_1 = H(\vec{z}^1)$ and $h_2 = H(-\vec{z}^2)$, where H is a collision-resistant

hash function, and $\vec{z}^i$ is the share of $\vec{z}$ held by V_i. Similarly to the broadcast that we defined in Section 4.3.1, if $h_1 = h_2$, it should with high probability be $\vec{z}^1 = -\vec{z}^2$, implying $\vec{z} = \vec{z}^1 + \vec{z}^2 = \vec{0}$.

A corrupted verifier may intentionally provide wrong h_i, $\hat{a}^i$, or $\hat{b}^i$, causing the correctness check to fail. It will not be clear whether P or V_i is guilty. Such failure is allowed by $\mathcal{F}_{pre}$ since it does not handle private data. Alternatively, all shares could be opened through $\mathcal{F}_{transmit}$ to identify the cheater.

If all $\mu - 1$ checks succeed, then the first $\mu - 1$ tuples in each group are discarded and only the last one is used. Since a pairwise check passes only if *both* tuples are incorrect, the corrupted prover needs to make *all* μ tuples in a group incorrect to make a single incorrect tuple accepted, and this probability is made negligible by adjusting the parameters μ and κ.

A combinatorial analysis, given in details in Section 4.5.4, shows that values μ and κ do not need to be large to bound the prover's cheating probability by 2^{-80}. For example, if $u = 2^{20}$, then it is sufficient to take $\mu = 5$ and $\kappa = 1300$. If $u = 2^{30}$ then $\mu = 4$ and $\kappa = 14500$ are sufficient. At the other extreme, if $u = 10$, then $\mu = 26$ and $\kappa = 168$ are sufficient for the same security level.

In a finite field, more efficient methods than cut-and-choose and pairwise check are available. For example, we can replace them with an application of linear error correcting codes [6]. This technique allows to construct u verified tuples from only $u + \kappa$ initial ones, where κ is proportional to the security parameter η.

4.3.3 Protocol Implementing $\mathcal{F}_{verify}$

The protocol Π_{verify} implementing $\mathcal{F}_{verify}$ is given in Figure 4.7-4.8. All communication between parties takes place using $\mathcal{F}_{transmit}$. In this way, if a party refuses to send a properly formatted message, it will be publicly blamed. If the prover is blamed, then its proof does not proceed further. If one of the verifiers is blamed, then the proofs of other parties may be halted, since they should be honest assuming at most one corrupted party. Hence, without loss of generality, we assume that all the transmissions of $\mathcal{F}_{transmit}$ succeed.

Initialization. The initialization fixes the circuits C_{ij}^{ℓ} that are going to be verified. A sufficient number of precomputed tuples is generated by $\mathcal{F}_{pre}$. The number of these tuples and their types depends on the gates of C_{ij}^{ℓ}, described more precisely in Figure 4.7. The verification phase clarifies why exactly these tuples are generated.

Randomness commitment. The prover P must fairly choose the (uniformly distributed) randomness it is going to use as the input of the composition of its circuits C_i, and commit to it. For this purpose, the *verifiers* jointly generate it.

• **Initialization:** The protocol starts with each party getting the input $(\mathsf{init}, (C_{ij}^{\ell})_{i,j,\ell}^{n,n,r})$, where the composition of C_{ij}^{ℓ} for each i is denoted C_i. The circuit defines the ring sizes $m(xid_k^i)$ of inputs and $m(yid_k^i)$ of outputs of C_i. As a shorthand notation, let $P = P_i$, $P' = P_j$. Let V_1, V_2 be the verifiers of P, and V_1', V_2' the verifiers of P'.
The subroutine $\mathcal{F}_{pre}$ is called to generate a sufficient number of precomputed tuples for each party. The number of tuples and their types depend on the gates of the circuits C_i.

1. *Linear combination, conversion to a smaller ring:* no tuples needed;
2. *Multiplication in $\mathbb{Z}_m$:* one multiplication triple over $\mathbb{Z}_m$;
3. *Bit decomposition in $\mathbb{Z}_{2^m}$:* m trusted bits over $\mathbb{Z}_{2^m}$;
4. *Conversion from $\mathbb{Z}_{2^n}$ to a larger ring $\mathbb{Z}_{2^m}$:* n trusted bits over $\mathbb{Z}_{2^m}$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, xid_k^i)$, V_1 generates $r_1 \xleftarrow{\$} \mathbb{Z}_m$, and V_2 generates $r_2 \xleftarrow{\$} \mathbb{Z}_m$. They send r_1 and r_2 to P using $\mathcal{F}_{transmit}$. On input $(\mathsf{commit\_rnd}, xid_k^i)$, P expects to receive r_1 and r_2 from $\mathcal{F}_{transmit}$. It takes $r = r_1 + r_2$. Now r is treated as the committed k-th input of C_i.

• **Input Commitment:** On input $(\mathsf{commit\_input}, x, xid_k^i)$, P shares $x = x^1 + x^2$ in $\mathbb{Z}_m$ and uses $\mathcal{F}_{transmit}$ to deliver x^1 to V_1 and x^2 to V_2. On input $(\mathsf{commit\_input}, xid_k^i)$, V_1 and V_2 expect to receive x^1 and x^2 respectively from $\mathcal{F}_{transmit}$. Now x is treated as the committed k-th input of C_i.

• **Message Commitment:**

1. On input $(\mathsf{send\_msg}, x, yid_l^i, xid_k^j)$, P uses $\mathcal{F}_{transmit}$ to deliver x to P'. On input $(\mathsf{send\_msg}, yid_l^i, xid_k^j)$, P' expects to receive x from $\mathcal{F}_{transmit}$.
2. On input $(\mathsf{commit\_msg}, yid_l^i, xid_k^j)$, the verifier $V_1 = P'$ takes the share $m^1 = m$, and the other verifier $V_2 \neq P'$ takes the share $m^2 = 0$. Analogously, treating P' as a prover, the verifier $V_1' = P$ takes the share $m^1 = m$, and the other verifier $V_2' \neq P$ takes the share $m^2 = 0$. Now m is treated as the committed l-th output of C_i and the k-th input of C_j.

Figure 4.7: Real protocol Π_{verify} (initialization and commitments)

• **Verification (1st round):** On input (verify, i), the prover P broadcasts some hints that will be used by V_1 and V_2 to localize their computation. These values depend on the gates of the circuit C_i.

1. *Linear combination, conversion to a smaller ring*. No broadcasts needed.
2. *Multiplication in* $\mathbb{Z}_m$. Let $[\![y]\!] = [\![x_1]\!] \cdot [\![x_2]\!]$ be verified. Let $([\![a]\!], [\![b]\!], [\![c]\!])$ be a precomputed multiplication triple over $\mathbb{Z}_m$. P broadcasts $\hat{x}_1 = x_1 - a$ and $\hat{x}_2 = x_2 - b$.
3. *Bit decomposition in* $\mathbb{Z}_{2^m}$. Let $([\![y_0]\!], \ldots, [\![y_{m-1}]\!]) = \mathsf{bd}([\![x]\!])$ be verified. Let $[\![b_0]\!], \ldots, [\![b_{m-1}]\!]$ be precomputed trusted bits, shared over $\mathbb{Z}_m$. P broadcasts bits $c_0, \ldots, c_{m-1}$, where $c_k = 1$ iff $b_k = y_k$.
4. *Conversion from* $\mathbb{Z}_{2^n}$ *to a larger ring* $\mathbb{Z}_{2^m}$. Let $y = \mathsf{zext}(x)$ be verified. Let $[\![b_0]\!], \ldots, [\![b_{n-1}]\!]$ be precomputed trusted bits, shared over $\mathbb{Z}_{2^m}$. P performs bit decomposition of x over $\mathbb{Z}_{2^m}$, getting m bits x_k. It takes the first n of these bits, and broadcasts $c_0, \ldots, c_{n-1}$, where $c_k = 1$ iff $b_k = x_k$.

• **Verification (2nd round):** After the broadcasts have been done, the verifiers start computing C_i locally on shares, collecting the alleged zeroes. V_1 and V_2 compute the gates as follows.

1. *Linear combination*. Let $y = \sum_{j=1}^{t} c_j \cdot x_j$ be verified. Compute $[\![y]\!] = \sum_{j=1}^{t} c_j \cdot [\![x_j]\!]$.
2. *Multiplication in* $\mathbb{Z}_m$. Using $\hat{x}_1$ and $\hat{x}_2$ that P has broadcast, compute $[\![y]\!] = \hat{x}_1 \cdot [\![x_2]\!] + \hat{x}_2 \cdot [\![a]\!] + [\![c]\!]$. Compute the alleged zeroes $[\![z_1]\!] = [\![x_1]\!] - [\![a]\!] - \hat{x}_1$ and $[\![z_2]\!] = [\![x_2]\!] - [\![b]\!] - \hat{x}_2$.
3. *Bit decomposition in* $\mathbb{Z}_{2^m}$. Using the bits c_k that P has broadcast, take $[\![y_k]\!] = [\![b_k]\!]$ if $c_k = 0$, and $[\![y_k]\!] = 1 - [\![b_k]\!]$ if $c_k = 1$. Compute the alleged zero $[\![z]\!] = [\![x]\!] - \sum_{i=0}^{m-1} 2^k \cdot [\![y_k]\!]$.
4. *Conversion from* $\mathbb{Z}_{2^n}$ *to a smaller ring* $\mathbb{Z}_{2^m}$. Drop $n - m$ highest bits from all shares of x.
5. *Conversion from* $\mathbb{Z}_{2^n}$ *to a larger ring* $\mathbb{Z}_{2^m}$: Perform the bit decomposition of $[\![x]\!]$, obtaining the shared bits $[\![y_0]\!], \ldots, [\![y_{n-1}]\!]$; the bits are shared over the ring $\mathbb{Z}_{2^m}$. Compute $[\![y]\!] = \sum_{i=0}^{n-1} 2^k \cdot [\![y_k]\!]$ and the alleged zero $[\![z]\!] = [\![x]\!] - \sum_{i=1}^{n-1} 2^k \cdot \mathsf{trunc}([\![y_k]\!])$.
6. *Circuit outputs*: Let $[\![y]\!]$ be the output locally computed by the verifiers. Let $[\![y']\!]$ be the output committed before. Compute the alleged zero $[\![z]\!] = [\![y]\!] - [\![y']\!]$.

V_1 computes $h_1 = H(z_1^1, z_2^1, \cdots, z_s^1)$ and V_2 computes $h_2 = H((-z_1^2), (-z_2^2), \cdots, (-z_s^2))$, where H is a collision-resistant hash function and z_k^1, z_k^2 are the shares of $[\![z_k]\!]$ held by the first and second verifier, respectively. They send h_1 and h_2 to each other and to the prover, checking if $h_1 = h_2$. If $h_1 \neq h_2$, then P may broadcast a complaint against one of the verifiers V_k. In this case, all shares of V_k are opened through $\mathcal{F}_{transmit}$, and V_j repeats the computation of V_k.

• **Cheater detection:** At any time, when $\mathcal{F}_{transmit}$ outputs a message $(\mathsf{cheater}, k)$, output $(\mathsf{cheater}, k)$ and stop.

Figure 4.8: Real protocol Π_{verify} (verification and cheater detection)

Each verifier V_j generates a uniformly distributed r_j and uses $\mathcal{F}_{transmit}$ to deliver r_j to P. After receiving r_1 and r_2, P takes the randomness $r = r_1 + r_2$ that is additively shared among V_1 and V_2. Since at least one verifier V_j is honest, and the other verifier does not know anything about the value r_j, the randomness r is distributed uniformly. In the security proof, the simulator is able to simulate exactly the same r that has been chosen by $\mathcal{F}_{verify}$, taking $r_j = r - r_i$ after the adversary has chosen r_i for the corrupted verifier.

Input commitment. At the beginning of protocol execution, P commits to its input x by sharing it as $x = x_1 + x_2$ and using $\mathcal{F}_{transmit}$ to deliver x_i to V_i. The share issued to the corrupted verifier is distributed uniformly and is easy to simulate. A corrupted prover may choose any x and share it in an arbitrary way. This is allowed by $\mathcal{F}_{verify}$.

Message commitment. During the protocol execution, the sender transmits each message m using $\mathcal{F}_{transmit}$. The sender can be the prover P as well as some other party P'. As the result, each message m that has been sent or received by P is known at least to one verifier V_1 or V_2 that has been on the other side of the communication. Since each such message m has been delivered using $\mathcal{F}_{transmit}$, it is possible to prove its authenticity later, and hence both the sender and the receiver have been committed to the same m. For both of them, it can be viewed as being additively shared among the verifiers as $m = m + 0$.

Verifying local computations. The local computation of P_i is represented by circuits C_{ij}^{ℓ} turning already received messages to new messages of the next round (see Section 4.2).

The circuits are verified gate-by-gate. For each gate, we have the following setup. The gate operation op takes k inputs in some ring $\mathbb{Z}_m$ and produces l outputs in some ring $\mathbb{Z}_{m'}$. The input values are shared as $[\![x_1]\!], \ldots, [\![x_k]\!]$ among the verifiers. The prover knows all these shares. During the computation of the circuit, the prover was expected to apply op to $x_1, \ldots, x_k$ and obtain the outputs $y_1, \ldots, y_l$. The verifiers are sure that the shares they have indeed correspond to $x_1, \ldots, x_k$ (subject to some deferred checks). A verification step gives us $[\![y_1]\!], \ldots, [\![y_l]\!]$ shared among the verifiers, where the prover again knows the shares of both verifiers, but no verifier has learned anything new. The verification step also gives us a number of alleged zeroes $[\![z_1]\!], \ldots, [\![z_s]\!]$, all known to the prover. If $z_1 = \cdots = z_s = 0$ then the verifiers are sure that the sharings $[\![y_1]\!], \ldots, [\![y_l]\!]$ indeed correspond to $y_1, \ldots, y_l$. All these equality checks are deferred to be succinctly verified one round later.

Repeating this process gate by gate, the verifiers finally obtain a sharing $[\![y]\!]$ of some output of the circuit from the commitments to its inputs. The prover has previously committed that output as $[\![y']\!]$ (the output is a message that the prover has sent to another party). To verify the correctness of prover's commitment, the parties produce an alleged zero $[\![z]\!] = [\![y]\!] - [\![y']\!]$.

For particular gate operations, the values $[\![y_i]\!]$ and $[\![z_i]\!]$ are computed as shown in Figure 4.7. First, the prover broadcasts to the verifiers some hints, which are just differences between private values and components of the precomputed tuples. Similarly to the pairwise check of Π_{pre}, since each tuple is used only once, all these values come from uniform distribution and can be easily simulated in the security proof. Using these hints and the precomputed tuples, all circuit operations can be reduced to linear combinations of shared values, computed using the homomorphic properties of the sharing scheme.

It is easy to check that, if $z = 0$ for all alleged zeroes z, then $(y_1, \ldots, y_l) = op(x_1, \ldots, x_k)$ for all gate operations op. The correctness of all broadcast hints is verified using alleged zeroes. The multiplication check is analogous to the pairwise check of Π_{pre}, and for the bit operations, since $y_i \in \{0, 1\}$ (it follows from $b_i \in \{0, 1\}$), the equality $[\![x]\!] = \sum_{i=0}^{m-1} 2^i \cdot [\![y_i]\!]$ implies that $(y_0, \ldots, y_{m-1})$ is an m-bit decomposition of x.

So far, all the communication between parties only originates from the prover. Thus the verification of a circuit can be done by the prover first broadcasting a single long message, followed by the verifiers performing local computations.

Checking of alleged zeroes. The verifiers check if $\vec{z} = (z_1, \ldots, z_s)$ is equal to $\vec{0}$ similarly to Π_{pre}, exchanging the hashes h_1 and h_2 of shares $\vec{z}^1$ and $(-\vec{z}^2)$.

If $h_1 \neq h_2$, it is possible that not P, but some V_k has cheated by publishing an incorrect h_k. In this case, h_1 and h_2 are also opened to P, who holds all the shares and hence knows how h_1 and h_2 should look like. P is allowed to complain against one of the verifiers V_k. All the shares of V_k are opened through $\mathcal{F}_{transmit}$. The other verifier V_j can now repeat the computation of V_k and check whether P or V_k was cheating. After this step, V_j knows exactly who the cheater was. Both honest parties now agree on the cheater's identity. Similarly to the conflict resolving of $\mathcal{F}_{transmit}$, opening these shares can be easily simulated in the UC model since if there is a conflict between P and V_k, then all these shares are already known to the adversary.

All communication in this step originates from the verifiers, unless there are complaints. All these messages can be transmitted in the same round. The whole post-execution phase, in the case of no complaints, only requires two rounds of communication. The broadcasts of hints take place in the first round while exchanging the hashes of alleged zero shares takes place during the second round.

4.4 Generalization to Verifiable n-Party SMC with an Honest Majority

Let the number of parties be n. We assume that the majority of parties is honest. We show that this allows us to use linear threshold secret sharing to make P and $V_1, \ldots, V_{n-1}$ (some of which may be corrupted) together act as an honest verifier. The largest challenge coming from $n > 3$ is that the corrupted prover P is now able to collaborate with some of the corrupted verifiers V_i.

In this section, we show how the building blocks of Section 4.3.1 can be generalized to n-party case. We also review the definitions of Π_{pre} and Π_{verify}, generalizing them to n parties.

4.4.1 Building Blocks

Ensuring message delivery. Assuming an honest majority, the functionality $\mathcal{F}_{transmit}$ of [29] works for any number n of parties. If both the sender and the receiver are corrupted, they are not bound to the transmitted messages, and may reveal anything afterwards. This is sufficient for our settings.

Broadcast and opening. Broadcast and opening can still be build on top of $\mathcal{F}_{transmit}$. Now *each* pair of parties P_i and P_j will use the hash-based consistency check to verify that they received the same message. If $h_i \neq h_j$, then m_i and m_j are revealed to all parties, publicly identifying the cheater. The ability of corrupted sender and a corrupted receiver to reveal any value just allows them to decide who of them will be blamed. If the sender is accused, then the broadcast fails. If the receiver is accused, then its hashes h_i are ignored by all parties. In the end, either the sender is accused by all honest parties, or each party has agreed on m with at least $t - 1$ other honest parties.

Sharing based commitments. The commitments can be done using any linearly homomorphic (n, t)-threshold sharing scheme. Formally, the prover P is treated as one of the share holders, but in practice P needs to come into play only after all $t - 1$ corrupted verifiers have been caught in cheating. All shares that P sends to V_i are delivered by $\mathcal{F}_{transmit}$. It prevents corrupted parties from tampering with the shares of honest provers, and prevents a corrupted prover from repudiating the shares that it has given to the honest verifiers.

If the number of honest parties is at least t, then there is a subset of t verifiers $\mathcal{H}$ that lists only honest parties. In this case, a set of shares can be reconstructed to at most one value. Even if corrupted verifiers collaborate with a corrupted prover and modify their shares later ($\mathcal{F}_{transmit}$ does not commit corrupted parties

to each other), this may only lead to inconsistency of shares, and failure to open the commitment. Availability of at least t honest parties allows to maintain the commitment even if all the corrupted parties have left the protocol.

Some examples of suitable secret sharing schemes are given in Section 2.4.2. Shamir's sharing is an example of (n, t)-threshold sharing that works over any finite field. For ring operations, replicated secret sharing can be used. We note that the size of shares in the latter case grows exponentially with n.

Preprocessed tuples. Using linear (n, t)-threshold sharing instead of additive, the ideal functionality $\mathcal{F}_{pre}$ can be directly generalized to n parties. Since the sharing scheme is still linear, all the steps of Π_{pre}, up to alleged zero check, can be repeated similarly to the 3-party case, without additional interaction. By properties of (n, t)-threshold sharing, either the shares of z (and also the opened $\hat{a}$ and $\hat{b}$) are inconsistent, or z is equal to the value that has been computed according to the protocol rules from the shares of $\mathcal{H}$. The only difference from the 3-party case is that the verifiers cannot simply exchange the hashes $h_1 = H(z_1^1, \ldots, z_s^1)$ and $h_2 = H((-z_1^2), \ldots, (-z_s^2))$. Instead, they need to broadcast z_j^i in plain. If the opened shares are inconsistent, the protocol aborts.

4.4.2 Generalization of Π_{verify}

In generalized protocol, all the commitments are done using linear (n, t)-threshold sharing instead of additive.

Input commitment. The shares are generated by the prover itself, similarly to the 3-party case. The consistency of shares is not being checked. The commitment is determined by the shares of $\mathcal{H}$ anyway, and it may be only more difficult for the prover to make its proof hold for inconsistent shares.

Randomness commitment. Each verifier V_j first generates $r_j \xleftarrow{\$} \mathbb{Z}_m$ and commits itself to it by sharing. After V_k has received the shares r_j^k of all the other verifiers V_j, it uses $\mathcal{F}_{transmit}$ to deliver r_j^k to P. After receiving all r_j^k, P reconstructs r_j, and takes $r = \sum_j r_j$. If the shares of some r_j are inconsistent, then all the shares are revealed through $\mathcal{F}_{transmit}$. The cheater is discarded, and the randomness commitment is restarted, this time without the cheater.

It is very important that V_k opens r_j^k to P only *after* it receives r_j^k from all V_j. This prevents a corrupted P from getting r_j of honest verifiers *before* all corrupted V_i have been committed to r_i that they generated, thus preventing corrupted V_i from making r_i dependent on r_j of honest V_j, keeping r uniformly distributed.

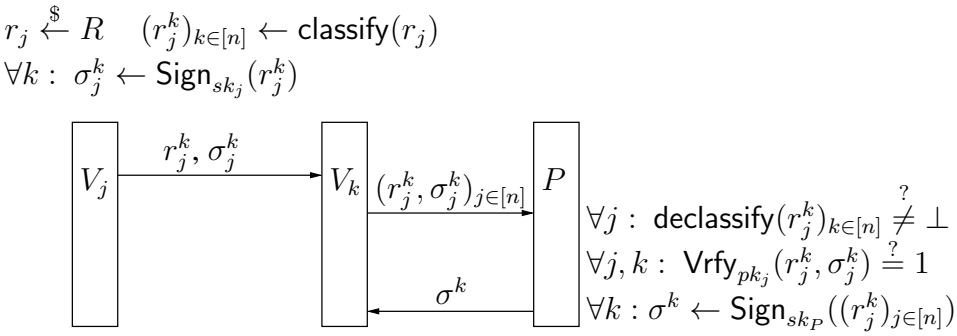


Figure 4.9: Committing to randomness in a ring R

Assuming that $\mathcal{F}_{transmit}$ has been implemented using signatures as shown in Figure 4.12, security proofs are easier if all commitments are confirmed with P's signatures. For this, it is sufficient to add one more round in which P confirms the shares by sending to V_k a signature on $(r_j^k)_{j\in[n]}$. The pictorial representation of resulting protocol, decomposed to the details of signature-based implementation of $\mathcal{F}_{transmit}$, is given in Figure 4.9. In this example, V_k uses the signatures of V_j instead of signing r_j^k itself to reduce the number of different signatures that have to be opened in case of cheating, and $\mathcal{F}_{transmit}$ is not used as a black box.

Message commitment. During the execution, all messages are transmitted using $\mathcal{F}_{transmit}$, as in the 3-party case. However, all messages now need to be additionally committed by sharing after the execution phase. For this, the sender P_s secret-shares the message m it had sent to some receiver P_r during the execution, and sends each share m^k to V_k using $\mathcal{F}_{transmit}$. All these shares are revealed through $\mathcal{F}_{transmit}$ to P_r who checks that m has been properly shared, and that it is the same m that it received in the execution phase. If the shares are inconsistent, or m is different, P_r is allowed to complain and open the message m that it has actually received from P_s in the execution phase. In this case, either P_s or P_r is corrupted, and the adversary already knows m that was actually transmitted.

Assuming that $\mathcal{F}_{transmit}$ has been implemented using signatures as shown in Figure 4.12, security proofs are easier if all commitments are confirmed with P_r's signatures. A straightforward solution would be to let P_r confirm the shares that it received by sending back to V_k the signatures on them. To avoid this additional round, we may instead assume that P_s sends all shares and their signatures directly to P_r who forwards them to V_k. The pictorial representation of this protocol is given in Figure 4.10. In this example, $\mathcal{F}_{transmit}$ is not used as a black box.

At this point, both P_s and P_r are committed to the shares of $[\![m]\!]$ that have been issued to the honest parties. It may happen that the sharing $[\![m]\!]$ does not

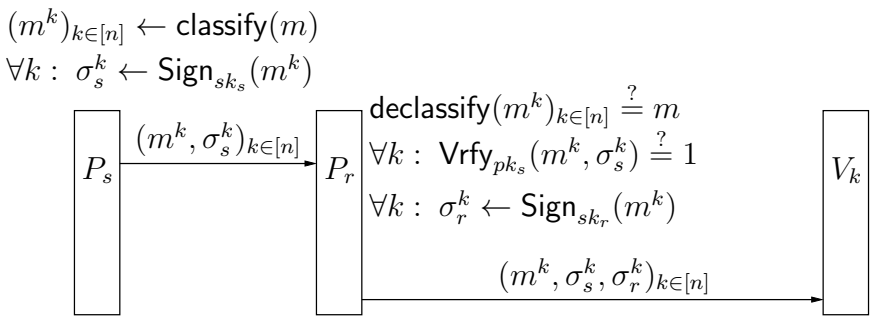


Figure 4.10: Committing to messages

correspond to the m transmitted in the execution phase only if P_s and P_r both are corrupted. In this case, the value of m that was actually transmitted is meaningless anyway, as it can be viewed as an inner value of the joint circuit of P_s and P_r. It is only important that P_s and P_r are committed to the same value. We recall that it is allowed by $\mathcal{F}_{verify}$.

Verification. The first round only involves some broadcasts by the prover, similarly to the 3-party case. On the second round, all the local computations can be done by the verifiers as in the 3-party case, since the linear (n, t)-threshold sharing has the necessary homomorphic property, and the sharing over a ring still allows to drop the highest bits of shares to get the same value shared in a smaller ring.

Similarly to Π_{pre}, the verifiers cannot use hashing to verify if $z = 0$, and they need to broadcast all shares of z instead. As in the 3-party case, the shares z^k do not leak any private information of an honest prover.

If $z = 0$, then it should be 0 also if we only take into account the shares z^k of $k \in \mathcal{H}$ that have honestly computed all the linear combinations w.r.t. the commitments. If $z \neq 0$, then it is not clear whether P or some verifier V_i has cheated (or both). In this case, P is allowed to complain about up to $t-1$ verifiers. All the shares of these verifiers are revealed through $\mathcal{F}_{transmit}$, and all the other verifiers repeat their proof steps to recompute their shares z^i. Similarly to the 3-party case, if there is a conflict, then all these shares are known to the adversary anyway, so they can be revealed.

Cheater detection. There are many steps in which a cheater can be detected due to use of $\mathcal{F}_{transmit}$. If one of the corrupted verifiers gets detected during the proof, then we still want the proof to finish, since it is not immediately clear whether P itself is a cheater. In all such cases, the corrupted verifier is discarded from the proof. Using (n, t)-threshold sharing allows the remaining parties to proceed with the proof, even after all $t - 1$ corrupted parties have left the protocol.

4.5 Security Proofs for n-Party Verifiable SMC with an Honest Majority

In this section, we formalize the protocols of Section 4.4 and give their security proofs. We use them to construct the protocol Π_{vmpc} UC-realizing $\mathcal{F}_{vmpc}$. We do not provide separate proofs for the 3-party protocols of Section 4.3, as they can be seen as instances of the n-party protocols. No new ideas are introduced in this section compared to Section 4.3-4.4, except Section 4.5.8 that presents a different approach to the verification, based on probabilistically checkable proofs described in Section 2.7. The proofs are done in the UC model described in Section 2.2.3.

Theorem 4.1. *Let n be the number of parties. Let $\mathcal{C}$ be the set of covertly corrupted parties, $|\mathcal{C}| < n/2$. Assuming that there is a signature scheme with probability of existential forgery δ, there exists a protocol Π_{vmpc} UC-realizing an r-round functionality $\mathcal{F}_{vmpc}$ with correctness error $\varepsilon \leq 6n^2(n+r+2)\cdot\delta + 2^{-\eta}$ for a security parameter η. If the initial protocol of $\mathcal{F}_{vmpc}$ has M_x, M_r, M_c, bits of inputs, randomness, and communication respectively, its circuits have N_b gates requiring bit decompositions, N_m multiplication gates, and its largest used ring has cardinality 2^m, then the resulting protocol Π_{vmpc} has at most $13+r$ rounds (of which 10 come from the preprocessing, 1 from the input commitment, and 2 from the verification), and the communication of different phases has the following upper bounds (let $N_g := N_b + N_m$, and sh_n the number of times the bit width of the value shared among n parties is smaller than the bit width of its one share).*

- *Preprocessing:* $\mathsf{sh}_n \cdot (4n^3\eta m(N_b m + 3N_m) + 3n^2 M_r) + o(n^3\eta m N_b)$.
- *Execution:* $\mathsf{sh}_n \cdot (n \cdot M_x + M_c) + o(rn^2)$.
- *Postprocessing:* $\mathsf{sh}_n \cdot (2n^3 N_g m + n^2 M_c) + o(n^2 N_g m)$.

If some corrupted party starts deviating from the protocol, the number of rounds may at most double, and the communication may increase at most $2n$ times.

The aim of this section is to prove Theorem 4.1. Throughout this section, we use $\mathcal{A}$ to denote the adversary that attacks a real protocol, and $\mathcal{A}_S$ the adversary that attacks an ideal functionality. For all ideal functionalities $\mathcal{F}$, we assume that the inputs of corrupted parties are delivered directly to $\mathcal{A}_S$, so that we do not need to write it out explicitly. For this reason, we often write that the simulator starts doing something *on input*, meaning the inputs of corrupted parties.

We often use an informal expression x *is chosen by* $\mathcal{A}_S$ in definitions of ideal functionalities, where a message of the form $(command, id, x)$ comes from a corrupted party. Formally, in such cases the ideal functionality $\mathcal{F}$ sends

to $\mathcal{A}_S$ a message $(\mathsf{arrived}(command), id, x)$, and waits until $\mathcal{A}_S$ sends back $(\mathsf{change}(command), id, x')$, so that $\mathcal{F}$ will further use x' instead of x. For shortness of presentation, we avoid writing out this sequence of messages.

Initially, the set of n parties executing a protocol is denoted by $[n]$. However, during the execution, it may happen that some parties will be discarded from the execution due to being detected in cheating. Therefore, we use the notation $\mathcal{P}$ to denote the set of parties that are currently active in the protocol execution.

In general, for all ideal functionalities and protocols of this section, at any time when a party P_k is detected in cheating, a message $(\mathsf{cheater}, k)$ is output to each party. In this case, P_k is discarded from the execution, resulting in $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$ (in the ideal functionality, also $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$). If the execution of some task ends up with outputting $(\mathsf{cheater}, k)$ to each party, it formally fails, but it can be immediately restarted without the cheater. The adversary is able to interrupt the execution at most $t-1$ times, until only honest parties $\mathcal{H}$ remain in the set $\mathcal{P}$. Since all our protocols are based on (n,t)-threshold sharing, they able to proceed with merely the set $\mathcal{H}$ of t honest parties.

4.5.1 Ensuring Message Delivery

In Figure 4.11, we give an extended version of message transmission functionality $\mathcal{F}_{transmit}$ that we first mentioned in Section 4.3.1. We include broadcast and public opening into the definition of $\mathcal{F}_{transmit}$, and we also allow to forward previously received messages. Each message is provided by a unique identifier id, encoding the sender $s(id)$ and the receiver $r(id)$ of this message, so that all parties know which messages need to be transmitted between which parties. It may also encode one more party $f(id)$ to which the message should be later forwarded by $r(id)$. For broadcasts, only $s(id)$ is important, since all parties of $\mathcal{P}$ are treated as receivers, and the values $r(id)$ and $f(id)$ may be undefined.

In contrast to the original definition of $\mathcal{F}_{transmit}$ of [29], we remove the requirement of synchronous delivery. This property ensures that the messages are delivered to the receivers only *after* all the messages have been sent by all senders of the given round. However, this property is hard to realize, since a corrupted sender may wait for messages of the other parties before sending its own messages. In our protocols, we only want to guarantee termination, making it possible to distinguish delayed messages from dropped messages. In order to achieve this kind of synchronicity, we may explicitly use e.g. Theorem 1 of [53] that proves feasibility of achieving synchronous computation in the UC model.

The protocol $\Pi_{transmit}$ implementing $\mathcal{F}_{transmit}$ is given in Figures 4.12-4.13. It works on top of signatures. Each message is signed by the sender, so that it can be revealed or forwarded afterwards. If the transmission fails, then the receiver broadcasts a complaint, and all other parties assist in the message delivery.

$\mathcal{F}_{transmit}$ works with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$, and a party $f(id) \in [n]$ to whom the message should be forwarded by the receiver (if no forwarding is foreseen then $f(id) = r(id)$, and for broadcasts the values of $r(id)$ and $f(id)$ do not matter).

• **Initialization:** On input $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$ from all (honest) parties, where $\hat{s}$,$\hat{r}$,$\hat{f}$ are mappings s.t $\mathsf{Dom}(\hat{s}) = \mathsf{Dom}(\hat{r}) = \mathsf{Dom}(\hat{f})$, assign $s \leftarrow \hat{s}$, $r \leftarrow \hat{r}$, $f \leftarrow \hat{f}$. Deliver (init, s, r, f) to $\mathcal{A}_S$.

• **Secure transmit:** On input $(\mathsf{transmit}, id, m)$ from $P_{s(id)}$ and $(\mathsf{transmit}, id)$ from all (honest) parties:

1. For $s(id) \in \mathcal{C}$, let m be chosen by $\mathcal{A}_S$.
2. Output (id, m) to $P_{r(id)}$, and $(id, |m|)$ to $\mathcal{A}_S$. If $r(id) \in \mathcal{C}$, output (id, m) to $\mathcal{A}_S$.
3. If $s(id) \in \mathcal{C}$, $\mathcal{A}_S$ may choose to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).

• **Broadcast:** On input $(\mathsf{broadcast}, id, m)$ from $P_{s(id)}$ and $(\mathsf{broadcast}, id)$ from all (honest) parties:

1. For $s(id) \in \mathcal{C}$, let m be chosen by $\mathcal{A}_S$.
2. Output (id, m) to each party and to $\mathcal{A}_S$.
3. If $s(id) \in \mathcal{C}$, $\mathcal{A}_S$ may choose to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).

• **Forward received message:** On input $(\mathsf{forward}, id)$ from $P_{r(id)}$ and on input $(\mathsf{forward}, id)$ from all (honest) parties, after (id, m) has been delivered to $P_{r(id)}$:

1. For $s(id), r(id) \in \mathcal{C}$, m is chosen by $\mathcal{A}_S$ instead of the value that was actually delivered.
2. Output (id, m) to $P_{f(id)}$, and $(id, |m|)$ to $\mathcal{A}_S$. If $f(id) \in \mathcal{C}$, output (id, m) to $\mathcal{A}_S$.
3. If $r(id) \in \mathcal{C}$, $\mathcal{A}_S$ may choose to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).

• **Reveal received message:** On input (reveal, id) from all (honest) parties, such that $P_{f(id)}$ at any point received (id, m), output (id, m) to each party, and also to $\mathcal{A}_S$.
If $s(id), r(id), f(id) \in \mathcal{C}$, then m is chosen by $\mathcal{A}_S$.
$\mathcal{A}_S$ may output $(\mathsf{cheater}, k)$ to all parties for any $k \in \mathcal{C} \cap \{s(id), r(id), f(id)\}$. If $(\mathsf{cheater}, k)$ is output for all $k \in \{s(id), r(id), f(id)\}$, then no (id, m) is output to the parties.

Figure 4.11: Ideal functionality $\mathcal{F}_{transmit}$

In $\Pi_{transmit}$, each party works locally with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$, and a party $f(id) \in [n]$ to whom the message should be forwarded by the receiver.

• **Initialization:** On input $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$, where $\mathsf{Dom}(\hat{s}) = \mathsf{Dom}(\hat{r}) = \mathsf{Dom}(\hat{f})$, each party assigns $s \leftarrow \hat{s}$, $r \leftarrow \hat{r}$, $f \leftarrow \hat{f}$. The parties exchange their public keys that will be used to verify signatures later.

• **Secure transmit:**

1. *Cheap mode:* use as far as $P_{r(id)}$ does not complain.
   (a) On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s. It sends (id, m, σ_s) to $P_{r(id)}$.
   (b) On input $(\mathsf{transmit}, id)$ the party $P_{r(id)}$ expects a message (id, m, σ_s) from $P_{s(id)}$, where σ_s is a valid signature from $P_{s(id)}$ on (id, m). If it receives it, it outputs (id, m) to $\mathcal{Z}$. If it does not receive it within one round, it broadcasts a signature $\gamma_{r(id)}$ on message $(\mathsf{bad}, s(id))$ using broadcast, and upon receiving it, each party goes to the expensive mode.
2. *Expensive mode:* an honest party goes to expensive mode if it receives a broadcast signature $\gamma_{r(id)}$ on $(\mathsf{bad}, s(id))$.
   (a) On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s. It sends (id, m, σ_s) to each other party.
   (b) Each party P_i sends (id, m, σ_s) to $P_{r(id)}$. If P_i does not receive (id, m, σ_s) within one round, it sends to $P_{r(id)}$ a signature γ_i on $(\mathsf{cheater}, s(id))$ instead.
   (c) On input $(\mathsf{transmit}, id)$, $P_{r(id)}$ expects a message (id, m, σ_s) from each P_i, where σ_s is a valid signature of $P_{s(id)}$ on (id, m). If it arrives from some P_i, then $P_{r(id)}$ outputs (id, m). Otherwise, $P_{r(id)}$ broadcasts all γ_i. Any party receiving γ_i from at least t parties outputs $(\mathsf{cheater}, s(id))$ to $\mathcal{Z}$.

• **Broadcast:**

1. On input $(\mathsf{broadcast}, id, m)$ the party $P_{s(id)}$ signs (id, m) to obtain signature σ_s and sends (id, m, σ_s) to each other party.
2. On input $(\mathsf{broadcast}, id)$ each party P_i expects a message (id, m, σ_s) from $P_{s(id)}$, where σ_s is a valid signature from $P_s(id)$ on (id, m). If no message arrives within one round, or the signature is invalid, it sends a signature γ_i on $(\mathsf{cheater}, s(id))$ to each other party. Otherwise, it sends the message (m, id, σ_s) to each other party. Any party receiving γ_i from at least t parties outputs $(\mathsf{cheater}, s(id))$ to $\mathcal{Z}$.
3. If any party receives (id, m, σ_s) and (id, m', σ'_s) for $m \neq m'$, it sends $(id, m, m', \sigma_s, \sigma'_s)$ to each other party. If indeed $m \neq m'$ and the signatures are valid, the honest party P_i receiving them outputs $(\mathsf{cheater}, s(id))$ to P_i. If P_i receives only messages (id, m, σ_s) with valid σ_s and no message (id, m', σ'_s) with $m \neq m'$ and valid σ'_s, then it outputs (id, m) to $\mathcal{Z}$.

Figure 4.12: Real Protocol $\Pi_{transmit}$ (secure transmission and broadcast)

- **Forward received message:**
  1. On input $(\mathsf{forward}, id)$ the party $P_{r(id)}$ that at some point received (id, m, σ_s) signs (id, m, σ_s) to obtain signature σ_r and sends $(id, m, \sigma_s, \sigma_r)$ to $P_{f(id)}$.
  2. On input $(\mathsf{forward}, id)$ the party $P_{f(id)}$ waits for one round and then expects a message $(id, m, \sigma_s, \sigma_r)$ from $P_{r(id)}$, where σ_s [resp. σ_r] is a valid signature from $P_{s(id)}$ [resp. $P_{r(id)}$] on (id, m). If $P_{f(id)}$ receives the message, it outputs (id, m) to $P_{f(id)}$. If it does not receive the message, it broadcasts a signature $\gamma_{f(id)}$ on message $(\mathsf{bad}, r(id))$ using broadcast, and upon receiving it, each party goes to the expensive mode that is analogous to the expensive mode of transmit.
- **Reveal received message:**
  1. On input (reveal, id), the party $P_{f(id)}$ which at any point should have received the message $(id, m, \sigma_s, \sigma_r)$, sends $(\mathsf{reveal}, id, m, \sigma_s, \sigma_r, \sigma_f)$ to each other party.
  2. Each party in turn sends the message to each other party. Several different messages with valid signatures are handled by an honest party in the same way as for the broadcast, and there are now 3 signatures instead of one. The parties that already hold some signatures on message under id may present them now. If only a single $(\mathsf{reveal}, id, m, \sigma_s, \sigma_r, \sigma_f)$ is received, an honest party P_i outputs (id, m).
  3. If $(\mathsf{cheater}, f(id))$ is output, then it is the turn for $P_{r(id)}$ to send $(\mathsf{reveal}, id, m, \sigma_s, \sigma_r)$ to all parties. If $(\mathsf{cheater}, r(id))$ is output, then it is the turn for $P_{s(id)}$ to send $(\mathsf{reveal}, id, m, \sigma_s)$ to all parties. If all attempts have failed, then all honest parties agree that $s(id), r(id), f(id) \in \mathcal{C}$, and the revealing fails.

Figure 4.13: Real Protocol $\Pi_{transmit}$ (forwarding and revealing messages)

The broadcast is based on sending the message to each other party, followed by each pair of parties exchanging the messages they received, checking whether they have received the same message. From the definition of $\Pi_{transmit}$, we can count the number of rounds and the communicated bits of different operations.

Observation 4.1. Let λ be the number of bits in a signature. The round and bit communication complexities of applying different functions of $\Pi_{transmit}$ to an N-bit message are given in Table 4.1. The costs of signatures γ_i on (bad, k) and $(\mathsf{cheater}, k)$, and the additional rounds of broadcast and reveal that take place after $(\mathsf{cheater}, k)$ has been output, are counted as *one-time overhead*, since each such overhead may happen only once for P_k. We have counted the additional third broadcast round only once for all broadcasts of reveal, since correct signatures on $m \neq m'$ immediately cause $(\mathsf{cheater}, k)$ to be output for all $k \in \{s(id), r(id), f(id)\}$.

We note that there should formally be reserved an additional "empty" round for the cheap mode. This would be a certain time span within which the parties are waiting for possible complaints, and that would be silent in the optimistic setting, when no one attempts to cheat. We claim that the accusations, if any, can be as well handled in the next round. If any transmit or forward operation of $\mathcal{F}_{transmit}$

Table 4.1: Costs of different functionalities of $\Pi_{transmit}$ applied to N-bit messages, using λ-bit signatures

functionality	rounds	communicated bits
Cheap mode (as far as all parties follow the protocol)		
transmit	1	$N + \lambda$
broadcast	2	$n(n-1) \cdot (N + \lambda)$
forward	1	$N + 2\lambda$
reveal	2	$n(n-1) \cdot (N + 3\lambda)$
Expensive mode (if some party deviates from the protocol)		
transmit	2	$2(n-1) \cdot (N + \lambda)$
forward	2	$2(n-1) \cdot (N + 2\lambda)$
One-time malicious overhead (happens at most once per party)		
transmit	1	$n(n-1) \cdot \lambda$
broadcast	1	$2(n-1)^2 \cdot (N + \lambda)$
forward	1	$n(n-1) \cdot \lambda$
reveal	5	$(2n(n-1) + 2(n-1)^2) \cdot (N + 2\lambda)$

ends up in a complaint, it means that some party has not received the message that it expected, or there are some missing or inconsistent signatures. However, it has no effect on the next round outputs of the other parties that have not presented any complaints on this round. If the complaint comes during broadcast or reveal, it may happen that some party immediately starts using the value it received, while some other party cannot proceed. In both cases, any satisfied party has been convinced that the value indeed originates from the sender (broadcast) or was indeed confirmed by the parties $s(id)$, $r(id)$, $f(id)$ (reveal) due to the signatures, so it would do the same computations anyway, even if there were no complaints from the other parties.

For simplicity, the further proofs are done in the setting as if all accusations had come already before the next round starts. Otherwise, there would be too many case distinctions, and the proofs would become more complicated. Also, introducing an empty round would not affect the bit communication.

Lemma 4.1. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$ and existence of signature scheme with probability of existential forgery δ, the protocol $\Pi_{transmit}$ UC-realizes $\mathcal{F}_{transmit}$ with correctness error $\varepsilon < N \cdot \delta$ and simulation error 0, where N is the total number of sent messages.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{transmit}$ described in Figure 4.14-4.15. The simulator runs a local copy of $\Pi_{transmit}$. It also generates signing and verification keys for all n parties, using a preagreed signature scheme.

• **Initialization:** $\mathcal{S}$ receives (init, s, r, f) from $\mathcal{F}_{transmit}$. It generates public and secret keys for honest the parties, and simulates them exchanging their public keys. The public and the secret keys of $k \in \mathcal{C}$ are chosen by $\mathcal{A}$.

• **Secure transmit:** On input $(\mathsf{transmit}, id, m)$ if $s(id) \in \mathcal{C}$, and $(\mathsf{transmit}, id)$ if $s(id) \notin \mathcal{C}$:

1. *Cheap mode:*

   (a) For $s(id) \in \mathcal{C}$, $\mathcal{S}$ receives m^* and σ_s^* from $\mathcal{A}$. In its local copy of $\Pi_{transmit}$, it simulates sending (id, m^*, σ_s^*) to $P_{r(id)}$. For $s(id), r(id) \notin \mathcal{C}$, $\mathcal{S}$ gets the message length $|m|$ from $\mathcal{F}_{transmit}$. This is needed to model the view of $\mathcal{A}$ on messages moving through secure point-to-point channels between the honest parties.

   (b) For $s(id) \in \mathcal{C}$ and $r(id) \notin \mathcal{C}$, if $\mathcal{A}$ decides that $s(id)$ sends an invalid message, then $\mathcal{S}$ simulates broadcasting a signature on (bad, id) by $P_{r(id)}$, and goes to the expensive mode. For $s(id) \notin \mathcal{C}$, $r(id) \in \mathcal{C}$, $\mathcal{S}$ receives a message (id, m) from $\mathcal{F}_{transmit}$. It creates a signature σ_m on m and simulates delivery of (id, m, σ_s) to $P_{r(id)}$.

2. *Expensive mode:*

   (a) $\mathcal{S}$ signs (id, m) to obtain signature σ_s. $\mathcal{S}$ knows m since if the expensive mode is entered, then either the sender or the receiver is corrupt. $\mathcal{S}$ simulates sending (id, m, σ_s) to each other party. If $s(id) \in \mathcal{C}$, it acts as $\mathcal{A}$ tells.

   (b) $\mathcal{S}$ simulates sending (id, m, σ_s) by P_i to $P_{r(id)}$, where $\mathcal{A}$ decides what to send for $i \in \mathcal{C}$. It simulates sending a signature on $(\mathsf{cheater}, s(id))$ if necessary.

   (c) $\mathcal{S}$ simulates the honest behaviour of $r(id) \notin \mathcal{C}$. For $r(id) \in \mathcal{C}$, it acts as $\mathcal{A}$ tells to $P_{r(id)}$, simulating the broadcast of $(\mathsf{cheater}, k)$ if necessary.

• **Forward received message:** On input $(\mathsf{forward}, id, m)$ if $r(id) \in \mathcal{C}$, and $(\mathsf{forward}, id)$ if $r(id) \notin \mathcal{C}$:

1. For $r(id) \notin \mathcal{C}$ and $f(id) \in \mathcal{C}$, $\mathcal{S}$ receives (id, m) from $\mathcal{F}_{transmit}$, generates the signatures σ_s, σ_r on m, and simulates sending $(id, m, \sigma_s, \sigma_r)$ to $P_{f(id)}$. For $r(id) \in \mathcal{C}$ and $f(id), s(id) \notin \mathcal{C}$, $\mathcal{S}$ should ensure delivery of m that was sent by $s(id)$ on some point. $\mathcal{A}$ may choose some $m^* \neq m$ to be forwarded, and the signatures σ_s^*, σ_r^* on m^*.

2. $\mathcal{S}$ simulates the behaviour of $P_{f(id)}$ as it did on input $(\mathsf{transmit}, id)$, going to the expensive mode if necessary.

• **Broadcast:** On input $(\mathsf{broadcast}, id, m)$ if $s(id) \in \mathcal{C}$, and $(\mathsf{broadcast}, id)$ if $s(id) \notin \mathcal{C}$:

1. For $s(id) \notin \mathcal{C}$, $\mathcal{S}$ receives (id, m) from $\mathcal{F}_{transmit}$ and generates a signature σ_m on m. For $s(id) \in \mathcal{C}$, $\mathcal{S}$ receives m^i and σ_s^i for all i from $\mathcal{A}$. It simulates sending (id, m, σ_s) (or (id, m^i, σ_s^i)) to P_i.

2-4 For the next broadcast rounds, $\mathcal{S}$ simulates the honest behaviour of all $i \notin \mathcal{C}$. For $i \in \mathcal{C}$, it acts as $\mathcal{A}$ tells to P_i, simulating the broadcast of $(\mathsf{cheater}, k)$ if necessary. All messages that are needed in simulation are already known to $\mathcal{S}$. If the broadcast eventually succeeds, and some m^* should be output by each party, then $\mathcal{S}$ outputs m^* to $\mathcal{F}_{transmit}$, so that the (id, m^*) is output to all parties by $\mathcal{F}_{transmit}$. If the broadcast fails, then $\mathcal{S}$ sends $(id, \perp)$ to $\mathcal{F}_{transmit}$, so that $(\mathsf{cheater}, s(id))$ would be output to all parties by $\mathcal{F}_{transmit}$.

Figure 4.14: Simulator $\mathcal{S}_{transmit}$ (initialization, transmissions, broadcast)

• **Reveal received message:** On input (reveal, id), revealing is simulated similarly to the broadcast. The only difference is that, if a message $(\mathsf{cheater}, k)$ should be output by all (honest) parties, then the broadcast is repeated with $r(id)$ and $s(id)$ as senders. $\mathcal{A}$ decides which messages will be chosen by $k \in \mathcal{C} \cap \{s(id), r(id), f(id)\}$. If $\{s(id), r(id), f(id)\}$ are all corrupt, it may choose valid signatures σ_s^*, σ_r^*, σ_f^* for any m^*.

- If the message was transmitted (or forwarded) in the expensive mode, then at least one honest party P_i already holds a signature σ_s (or σ_r) on m that was transmitted before with at least one signature. If $\mathcal{A}$ chooses $m \neq m^*$, then P_i may present two valid signatures σ_s and σ_s^* (σ_r and σ_r^*) on two different messages.
- If the message was transmitted in the cheap mode, then no other parties may present contradictory signatures. In this case, the revealing of m^* succeeds, and $\mathcal{S}$ delivers m^* to $\mathcal{F}_{transmit}$, so that (id, m^*) is output to all (honest) parties.

Figure 4.15: Simulator $\mathcal{S}_{transmit}$ (revealing transmitted messages)

Simulatability. In $\Pi_{transmit}$, the real adversary $\mathcal{A}$ needs to get all the messages received by the corrupted parties. Any message m that is sent to a corrupted party is delivered by $\mathcal{F}_{transmit}$ to $\mathcal{S}$. It is sufficient to know the message length $|m|$ to simulate secure channels between honest parties.

For the additional rounds in the expensive mode, $\mathcal{S}$ needs the message m to simulate resolving the conflict (i.e. simulating all the other parties assisting in delivery of m). In this case, the value m is known to $\mathcal{S}$ since the expensive mode is entered if either $s(id) \in \mathcal{C}$ or $r(id) \in \mathcal{C}$. In the first case, m is chosen by $\mathcal{A}$. In the latter case, a message (id, m) comes from $\mathcal{F}_{transmit}$. In addition, $\mathcal{S}$ needs to generate the signatures of honest parties on messages m that is receives from $\mathcal{F}_{transmit}$, which is possible since $\mathcal{S}$ has instantiated the signature scheme itself.

Broadcasts and revealings are easy to simulate, since the message m is given to $\mathcal{S}$ by $\mathcal{F}_{transmit}$ in both cases. For messages moving between the honest parties, $\mathcal{S}$ computes $|m|$ directly from m to simulate secure point-to-point channels.

Correctness. We discuss the correctness of different modes. Since $\mathcal{S}$ does not have control over messages $(\mathsf{cheater}, k)$ that are output by $\mathcal{F}_{transmit}$, we need to ensure that $\mathcal{F}_{transmit}$ outputs $(\mathsf{cheater}, k)$ to all (honest) parties iff $\mathcal{S}$ simulates the same in $\Pi_{transmit}$.

- *Transmission (cheap):* As far as all the parties provide valid messages and signatures, the messages in simulated $\Pi_{transmit}$ are delivered in the same way as in $\mathcal{F}_{transmit}$.
- *Broadcast:* We need to ensure that, either each honest party gets the same message m, or all of them output $(\mathsf{cheater}, s(id))$. Suppose that $s(id)$ has sent a message $(id, m_i, \sigma^i_{s(id)})$ to the party P_i, for all $i \in [n]$. If P_i does not receive a valid message, it sends a signature γ_i on $(\mathsf{cheater}, s(id))$ to each other party. Otherwise, it delivers $(id, m_i, \sigma^i_{s(id)})$ to each other party.

If at least one honest party received a proper $(id, m_i, \sigma^i_{s(id)})$, then all honest parties get it. If no honest party receives it, then each (honest) party gets at least t complaints γ_i (including itself), so it outputs $(\mathsf{cheater}, s(id))$.

If any party receives (id, m, σ_s) and (id, m', σ'_s) for $m \neq m'$, it sends $(id, m, m', \sigma_s, \sigma'_s)$ to each other party, proving that $P_{s(id)}$ misbehaved. This situation is possible only if $P_{s(id)}$ has itself generated the contradictory signatures σ_s and σ'_s. Since the signature includes not only the message, but also the current protocol session and the message identifier id, there is no way for $\mathcal{A}$ to take signatures of some previous rounds or sessions. By properties of the signature scheme, if $s(id) \notin \mathcal{C}$, then $\mathcal{A}$ may succeed in generating σ_s and σ'_s for $m' \neq m$ with probability at most δ. Hence $s(id) \notin \mathcal{C}$ will be accused only with probability at most δ. If no $(id, m, m', \sigma_s, \sigma'_s)$ has been sent for $m \neq m'$, then all honest parties should have obtained the same message (id, m, id).

- *Transmission (expensive):* If something goes wrong, a signature on message (bad, id) will be broadcast to each party. We have just proven that, either all honest parties receive (bad, id), or they output $(\mathsf{cheater}, r(id))$ if the broadcast fails. In either case, each party P_i now expects (id, m_i, σ^i_s) from $P_{s(id)}$, and it forwards the received (id, m_i, σ^i_s) to $P_{r(id)}$, sending $(\mathsf{cheater}, s(id))$ instead if the signature is invalid (similarly to the broadcast). Differently from broadcast, if P_i gets two properly signed, but different messages $m \neq m'$, it does not distribute $(id, m, m', \sigma_s, \sigma'_s)$ to prove that $P_{s(id)}$ is malicious, but just proceeds with either m or m'. This is allowed since in $\mathcal{F}_{transmit}$ the message m for $s(id) \in \mathcal{C}$ is chosen by $\mathcal{A}_S$ anyway, and $\mathcal{S}$ may deliver to $\mathcal{F}_{transmit}$ the m that $P_{r(id)}$ would choose. If P_i gets no proper (id, m_i, σ^i_s), then it should have received at least t complaints of the honest parties, so all of them can now be broadcast to all parties. Each γ_i can be falsified with probability at most δ.

- *Forwarding:* A party $P_{r(id)}$ that already holds a signature σ_s on m creates one more signature σ_r on m. sending the message to $P_{f(id)}$. If both $s(id), r(id) \in \mathcal{C}$, then they may choose a new message m^* and create arbitrary signatures on it. This is allowed by $\mathcal{F}_{transmit}$. If $s(id) \notin \mathcal{C}$, then $r(id)$ may generate a signature σ^*_s on some other message m^* with probability at most δ. If $r(id) \notin \mathcal{C}$, then we would not reach forwarding unless σ_s would be a valid message on m. Hence $P_{f(id)}$ gets valid signatures on m only if it is the same m that was transmitted by $P_{s(id)}$ to $P_{r(id)}$, or otherwise the expensive mode is run for forwarding m.

- *Revealing messages:* We need to show that a correct m is output, unless $s(id), r(id), f(id) \in \mathcal{C}$. The party $P_{f(id)}$ sends to each other party a

message $(\mathsf{reveal}, id, m, \sigma_s, \sigma_r, \sigma_f)$. Similarly to broadcast, the message m is accepted by an honest party iff all three signatures σ_s, σ_r, σ_f correspond to m, and there is no $m' \neq m$ that is also provided with valid signatures.

If $s(id), r(id), f(id) \in \mathcal{C}$, then $\mathcal{F}_{transmit}$ allows to reveal any value. If at least one of them is honest, then its signature can be falsified with probability at most δ. If the broadcast just fails, the correct message m with proper signatures will be revealed either by $f(id) \notin \mathcal{C}$, or $r(id) \notin \mathcal{C}$.

As a summary, for each message identifier id, if $\mathcal{A}$ wants to force $m' \neq m$ to be delivered for $s(id) \notin \mathcal{C}$ [or $r(id) \notin \mathcal{C}$ in the case of forwarding], it should falsify at least the signature of $P_{s(id)}$ [$P_{r(id)}$] on m, which happens with probability at most δ. Alternatively, if $\mathcal{A}$ just wants to cause the honest parties to blame an innocent $P_{s(id)}$, then it should generate another message m' s.t $m \neq m'$, and $\sigma_{m'}$ is a valid signature of $P_{s(id)}$ on m', which also happens with probability at most δ. If the total number of transmitted messages is N, the probability of cheating in at least one of them is at most $N \cdot \delta$. □

Parallelization. If several messages need to be transmitted to the same party in the same round, it is enough to provide just one signature for all of them. The only problem is that, only some of these messages may need to be forwarded or revealed afterwards, and it should be possible to verify if the signature corresponds to that particular message. We note that the signature covering all the messages of one round can be efficiently constructed by computing a Merkle hash tree of the single signatures of all these messages [78]. If the signature should be verified for only one message, it is necessary to reveal the authentication path of that message, which is just taking one node from each level of the tree, and also the one-time public/private key pair for that particular message. In this way, instead of sending n signatures for n messages, it suffices to send just $\lceil \log n \rceil + 3$ signatures.

4.5.2 Linearly Homomorphic Commitments

We define a functionality $\mathcal{F}_{commit}$ that we use as a black box for storing commitments, computing their linear combinations, and opening them. It is given in Figure 4.16. Its implementation can be built it on top of any linearly homomorphic commitment scheme, not necessarily on sharing. In addition to ordinary commitments, it allows to perform *mutual commitments*, resulting in two parties being committed to the same value. The parties may compute various linear combinations of the commitments and open them. The *private opening* allows to open the value to one party, making that party committed to it. Although in this thesis we only need its weak version (that may fail), it can be extended to the strong version, similarly to the public opening.

Each committed value, as well as every linear combination computed from the commitments, is identified by a unique id. The committed values are stored in an array $comm$ as $comm[id]$. A new session of $\mathcal{F}_{commit}$ is initialized when it gets an input (init, m, p) from all (honest) parties, where the quantities m, p are mappings such that $m(id)$ is the bit size of the ring in which the value $comm[id]$ is committed, and $p(id)$ is the party committed to $comm[id]$. In order to simplify the initialization of $\mathcal{F}_{commit}$ when it will be called by outer protocols, these mappings have to be initialized only for the commitments, and they will be extended later to linear combinations and truncations computed from the commitments. In order to make the extensions uniquely determined, we allow to compute linear combinations only from the values for which $p(id)$ is the same.

It is possible that execution of some task of $\mathcal{F}_{commit}$ does not succeed. If it happens, then $(\mathsf{cheater}, k)$ is output to each party, where k is the deviator responsible for the failure. In this case, P_k is discarded from the execution, resulting in $\mathcal{F}_{commit}$ assigning $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$ and $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$. All tasks of $\mathcal{F}_{commit}$ are still applicable to the commitments of the remaining parties, i.e. such that $p(id), p(id') \in \mathcal{P}$.

The protocol Π_{commit} (Figure 4.17-4.18) implementing $\mathcal{F}_{commit}$ works on top of the message transmission functionality $\mathcal{F}_{transmit}$ defined in Section 4.5.1. It uses a linear (n, t)-threshold secret sharing scheme with $t = \lceil n/2 \rceil + 1$. The protocol description demonstrates why we need two different types of public opening. The *weak opening* (weak_open) is cheap, but it may fail, resulting in a set of suspects $\mathcal{K}$ whose guilt has not been proven yet. In the case of failure, the parties may run *strong opening* (open), which is expensive, but allows all honest parties to identify the parties of $\mathcal{K}$ that have actually cheated, and discard them from the protocol. As the result, either the commitment of $P_{p(id)}$ is finally opened, or $P_{p(id)}$ is publicly blamed by all honest parties, if it turns out that no party in $\mathcal{K}$ has cheated. Another possibility to detect cheaters comes from $\mathcal{F}_{transmit}$ that may output messages $(\mathsf{cheater}, k)$ to the parties. All detected cheaters are discarded from the active set of parties $\mathcal{P}$. It does not interrupt the work of Π_{commit}, since there still remain at least t honest parties able to reconstruct any sharing.

From the definition of Π_{commit}, we count the number of $\mathcal{F}_{transmit}$ operations being called for different functions. This allows us to estimate the round and the communication complexity based on the implementation of $\mathcal{F}_{transmit}$.

Observation 4.2. The number of $\mathcal{F}_{transmit}$ operations for applying different functions of Π_{commit} to an N-bit input (in the optimistic mode) is given in Table 4.2, where tr_M, bc_M, fwd_M, rev_M denote the *costs* (the number of rounds and the bit communication, as defined in Section 2.3) of transmit, broadcast, forward, reveal respectively on an M-bit message, and sh_n is the number of times the bit width the value shared among n parties is smaller than the bit width of its one share.

$\mathcal{F}_{commit}$ works with unique identifiers id, encoding the bit size $m(id)$ of the ring in which the value is committed, and the party $p(id)$ committed to the value. The commitments are stored in an array $comm$, and their derivations in an array $deriv$. For each id, the term $deriv[id]$ is a tree whose leaves are the initial commitments, and the inner nodes are lc, trunc operations applied to them. For the initially committed values, $deriv[id] = id$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$, define the mappings $m \leftarrow \hat{m}$, $p \leftarrow \hat{p}$. Deliver (init, m, p) to $\mathcal{A}_S$.

• **Extension:** On input $(\mathsf{ext}, \hat{m}, \hat{p})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$ and $\mathsf{Dom}(\hat{m}) \cap \mathsf{Dom}(m) = \emptyset$, extend the mappings $m \leftarrow m \cup \hat{m}$, $p \leftarrow p \cup \hat{p}$ over the new domain $\mathsf{Dom}(\hat{m}) \cup \mathsf{Dom}(m)$. Deliver $(\mathsf{ext}, \hat{m}, \hat{p})$ to $\mathcal{A}_S$.

• **Public Commit:** On input $(\mathsf{pcommit}, id, x)$ from all (honest) parties, write $comm[id] \leftarrow x$, and output $(\mathsf{confirmed}, id)$ to all parties. Output x to $\mathcal{A}_S$.

• **Commit:** On input (commit, id, x) from $P_{p(id)}$ and (commit, id) from all (honest) parties, write $comm[id] \leftarrow x$, and output $(\mathsf{confirmed}, id)$ to all parties. If $p(id) \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$, who may alternatively tell $\mathcal{F}_{commit}$ to output a message $(\mathsf{cheater}, p(id))$ to all parties.

• **Mutual Commit:** On input $(\mathsf{mcommit}, id, id', x)$ from $P_{p(id)}$, $(\mathsf{mcommit}, id, id', x')$ from $P_{p(id')}$, and $(\mathsf{mcommit}, id, id')$ from all (honest) parties, compare x and x'. If $x = x'$, then write $comm[id], comm[id'] \leftarrow x$, and output $(\mathsf{confirmed}, id, id')$ to all parties. If $x \neq x'$, output $(id, id', \perp)$ to all parties and to $\mathcal{A}_S$.
If $p(id) \in \mathcal{C}$ [resp. $p(id') \in \mathcal{C}$], then $\mathcal{A}_S$ chooses x [resp. x']. Alternatively, it may deliver to $\mathcal{F}_{commit}$ a message $(\mathsf{cheater}, p(id))$ or $(\mathsf{cheater}, p(id'))$ that is output by $\mathcal{F}_{commit}$ to all parties.

• **Compute Linear Combination:** On input $(\mathsf{lc}, \vec{c}, \vec{id}, id')$ from all (honest) parties, where $|\vec{c}| = |\vec{id}| =: \ell$, $id' \notin \mathsf{Dom}(p)$, and $p' = p(id_i)$ are the same for all $i \in \{1, \ldots, \ell\}$, let $m' \leftarrow \min(\{m(id_i) \mid i \in \{1, \ldots, \ell\}\})$:

1. Compute $y \leftarrow (\sum_{i=1}^{\ell} c_i \cdot comm[id_i]) \bmod 2^{m'}$.
2. Write $comm[id'] \leftarrow y$;
3. Assign $m(id') \leftarrow m'$, $p(id') \leftarrow p'$, $deriv[id'] \leftarrow \mathsf{lc}(\vec{c}, \vec{id})$.

Syntactic sugar: we write $(id' = \sum_{i=1}^{\ell} c_i \cdot id_i)$ instead of $(\mathsf{lc}, \vec{c}, \vec{id}, id')$.

• **Compute Truncation:** On input $(\mathsf{trunc}, m', id, id')$ from all (honest) parties, where $m(id) \geq m' \in \mathbb{N}$, and $id' \notin \mathsf{Dom}(p)$:

1. Compute $y \leftarrow comm[id] \bmod 2^{m'}$.
2. Write $comm[id'] \leftarrow y$;
3. Assign $m(id') \leftarrow m'$, $p(id') \leftarrow p(id)$, $deriv[id'] \leftarrow \mathsf{trunc}(m', id)$.

Syntactic sugar: we write $(id' = id \bmod 2^{m'})$ instead of $(\mathsf{trunc}, m', id, id')$.

• **Weak Open:** On input $(\mathsf{weak\_open}, id)$ from all (honest) parties, if $|\mathcal{C}| > 0$, output $comm[id]$ to $\mathcal{A}_S$, who decides whether $(id, comm[id])$ or $(id, \perp)$ is output to each party.

• **Open:** On input (open, id) from all (honest) parties, output $comm[id]$ to $\mathcal{A}_S$. If $p(id) \in \mathcal{C}$, it chooses whether $(id, comm[id])$ or $(\mathsf{cheater}, p(id))$ is output to each party.

• **Privately Open:** On input $(\mathsf{priv\_open}, id, id')$ from all (honest) parties, if $deriv[id] \neq id$, do nothing. Otherwise, write $comm[id'] = comm[id]$, output $(id, id', comm[id])$ to $P_{p(id')}$, and output $(\mathsf{confirmed}, id, id')$ to all parties. If $p(id') \in \mathcal{C}$, output $(id, id', comm[id])$ to $\mathcal{A}_S$. If $p(id) \in \mathcal{C}$, $\mathcal{A}_S$ may tell $\mathcal{F}_{commit}$ to output $(id, id', \perp)$ to each party instead.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ is output to all parties, do not accept any inputs including id s.t $p(id) = k$ anymore. Let $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$, $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$.

Figure 4.16: Ideal functionality $\mathcal{F}_{commit}$

In Π_{commit}, each party works locally with unique identifiers id, encoding the bit size $m(id)$ of the ring in which the value is shared, and the party $p(id)$ committed to the value. The parties use a linear (n,t)-threshold sharing scheme with $t = \lceil n/2 \rceil + 1$. Each party stores its own local copy of arrays $comm^k$ for $k \in [n]$, into which it writes the shares known to it. Each party stores a term $deriv[id]$ (represented by a tree whose leaves are the initial commitments, and the inner nodes are lc, trunc operations applied to them) to remember in which way each $comm^k[id]$ has been computed. For the initially committed values, let $deriv[id] = id$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$ where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$, each party, assigns $m \leftarrow \hat{m}$, $p \leftarrow \hat{p}$. It defines mappings s, r, and f, such that $s(id^k_{k'}) \leftarrow p(id)$, $r(id^k_{k'}) \leftarrow k$, and $f(id^k_{k'}) \leftarrow k'$, for all $id \in \mathsf{Dom}(p)$, $k, k' \in [n]$. In addition, it defines the senders $s(id^{\mathsf{bc}}_k) \leftarrow p(id)$ for the broadcasts (used for share opening). It sends (init, s, r, f) to $\mathcal{F}_{transmit}$.

• **Extension:** On input $(\mathsf{ext}, \hat{m}, \hat{p})$ from all parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$ and $\mathsf{Dom}(\hat{m}) \cap \mathsf{Dom}(m) = \emptyset$, extend the mappings $m \leftarrow m \cup \hat{m}$, $p \leftarrow p \cup \hat{p}$. For the new identifiers id, initialize a new instance of $\mathcal{F}_{transmit}$, similarly to the initialization.

• **Cheater detection:** At any time when a party receives $(\mathsf{cheater}, k)$ from $\mathcal{F}_{transmit}$, it outputs $(\mathsf{cheater}, k)$ to $\mathcal{Z}$. After outputting $(\mathsf{cheater}, k)$ to $\mathcal{Z}$, it does not accept any inputs including id s.t $p(id) = k$ anymore, and treats P_k as if it has left the protocol, i.e. $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

• **Public Commit:** On input $(\mathsf{pcommit}, id, x)$, each party writes $comm^k[id] \leftarrow x^k$ for all $k \in [n]$, where $(x^k)_{k\in[n]} = \mathsf{classify}(x)$. It outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Commit:**

1. On input (commit, id, x), $P_{p(id)}$ shares $(x^k)_{k\in[n]} = \mathsf{classify}(x)$. It sends $(\mathsf{transmit}, id^k_k, x^k)$ to $\mathcal{F}_{transmit}$, for all $k \in [n]$.
2. On input (commit, id), P_k waits until (id^k_k, x^k) comes from $\mathcal{F}_{transmit}$.

If all transmissions succeed, each party P_k writes $comm^k[id] \leftarrow x^k$, $deriv[id] \leftarrow id$, and outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$. Otherwise, if $(\mathsf{cheater}, p(id))$ comes from $\mathcal{F}_{transmit}$, each party outputs $(\mathsf{cheater}, p(id))$ to $\mathcal{Z}$.

• **Mutual Commit:**

1. On input $(\mathsf{mcommit}, id, id', x)$, $P_{p(id)}$ shares $(x^k)_{k\in[n]} = \mathsf{classify}(x)$. It sends $(\mathsf{transmit}, id^{p(id')}_k, x^k)$ to $\mathcal{F}_{transmit}$ for all $k \in [n]$.
2. On input $(\mathsf{mcommit}, id, id', x')$, $P_{p(id')}$ waits until $(id^{p(id')}_k, x^k)$ comes from $\mathcal{F}_{transmit}$. Upon receiving it for all $k \in [n]$, $P_{p(id')}$ checks if x^k are valid consistent shares, and if $x' = \mathsf{declassify}(x^k)_{k\in[n]}$. If the check does not pass, or the messages do not come, $P_{p(id')}$ broadcasts (bad, id, id'). Otherwise, $P_{p(id')}$ sends $(\mathsf{forward}, id^{p(id')}_k)$ to $\mathcal{F}_{transmit}$ for each $k \in [n]$.
3. On input $(\mathsf{mcommit}, id, id')$, P_k waits until $(id^{p(id')}_k, x^k)$ comes from $\mathcal{F}_{transmit}$.

If all transmissions and forwarding succeed, each party P_k writes $comm^k[id] = comm^k[id'] \leftarrow x^k$, $deriv[id] = deriv[id'] \leftarrow id$, and outputs $(\mathsf{confirmed}, id, id')$ to $\mathcal{Z}$. If $(\mathsf{cheater}, p(id))$ [resp. $(\mathsf{cheater}, p(id'))$] comes from $\mathcal{F}_{transmit}$, each party outputs it to $\mathcal{Z}$. If (bad, id, id') is broadcast by $P_{p(id')}$, each party outputs $(id, id', \perp)$ to $\mathcal{Z}$.

Figure 4.17: Real Protocol Π_{commit} (init, cheater detection, commitments)

• **Compute Linear Combination:** On input $(\mathsf{lc}, \vec{c}, \vec{id}, id')$, where $|\vec{c}| = |\vec{id}| =: \ell$, $id' \notin \mathsf{Dom}(p)$, and $p' = p(id_i)$ are the same for all $i \in \{1, \ldots, \ell\}$, for $m' \leftarrow \min(\{m(id_i) \mid i \in \{1, \ldots, \ell\}\})$, each party P_k

1. computes $y^k \leftarrow (\sum_{i=1}^{\ell} c_i \cdot comm^k[id_i]) \bmod 2^{m'}$ ($P_{p(id)}$ computes all $(y^k)_{k\in[n]}$);
2. writes $comm^k[id'] \leftarrow y^k$ ($P_{p(id)}$ does it for all $k \in [n]$);
3. assigns $m(id') \leftarrow m'$, $p(id') \leftarrow p'$, $deriv[id'] \leftarrow \mathsf{lc}(\vec{c}, \vec{id})$.

• **Compute Truncation:** On input $(\mathsf{trunc}, m', id, id')$, where $m(id) \geq m' \in \mathbb{N}$, and $id' \notin \mathsf{Dom}(p)$, each party P_k

1. computes $y^k \leftarrow comm^k[id] \bmod 2^{m'}$ ($P_{p(id)}$ computes all $(y^k)_{k\in[n]}$);
2. writes $comm^k[id'] \leftarrow y^k$ ($P_{p(id)}$ does it for all $k \in [n]$);
3. assigns $m(id') \leftarrow m'$, $p(id') \leftarrow p(id)$, $deriv[id'] \leftarrow \mathsf{trunc}(m', id)$.

• **Weak Open:** On input $(\mathsf{weak\_open}, id)$:

1. $P_{p(id)}$ takes $x^k \leftarrow comm^k[id]$ and sends $(\mathsf{broadcast}, id_k^{\mathsf{bc}}, x^k)$ to $\mathcal{F}_{transmit}$ for all $k \in [n]$.
2. Upon receiving all shares $(id_k^{\mathsf{bc}}, x^k)$ from $\mathcal{F}_{transmit}$, each party P_j compares x^j with the share $comm^j[id]$ that it holds. If $x^j \neq comm^j[id]$, it broadcasts (bad, id).

If the shares are inconsistent, or some of them do not arrive, each party outputs $(\mathsf{cheater}, p(id))$ to $\mathcal{Z}$. If at least one message (bad, id) has been broadcast, each party constructs a set $\mathcal{K} := \{k \mid (\mathsf{bad}, id) \text{ was broadcast by } P_k\}$. If $\mathcal{K} \geq t$, then each party outputs $(\mathsf{cheater}, p(id))$ to $\mathcal{Z}$, and otherwise it outputs $(id, \perp)$ to $\mathcal{Z}$. If there are no problems, then each honest party reconstructs $x \leftarrow \mathsf{declassify}(x^k)_{k\in[n]}$ and outputs x to $\mathcal{Z}$.

• **Open:** On input (open, id), called after $(\mathsf{weak\_open}, id)$ has failed:

1. Each party sends $(\mathsf{reveal}, id_j^k)$ to $\mathcal{F}_{transmit}$ for all leaf identifiers id_j^k of the derivation term $deriv[id]$, for $k \in \mathcal{K}$.
2. Upon receiving all (id_j^k, x_j^k) from $\mathcal{F}_{transmit}$ for all $k \in \mathcal{K}$, using them to locally recompute the share x^k of $deriv[id]$, and taking x^k broadcast during $\mathsf{weak\_open}$ for $k \notin \mathcal{K}$, each party reconstructs $x \leftarrow \mathsf{declassify}(x^k)_{k\in[n]}$, and outputs x to $\mathcal{Z}$. If x^k are inconsistent, output $(\mathsf{cheater}, p(id))$ to $\mathcal{Z}$.

• **Privately Open:** On input $(\mathsf{priv\_open}, id, id')$, if $deriv[id] = id$:

1. Each party P_k sends $(\mathsf{forward}, id_{p(id')}^k)$ to $\mathcal{F}_{transmit}$.
2. Upon receiving all $(id_{p(id')}^k, x^k)$ from $\mathcal{F}_{transmit}$, $P_{p(id')}$ reconstructs $x \leftarrow \mathsf{declassify}(x^k)_{k\in[n]}$ and sends $(\mathsf{transmit}, id'^k_k, x^k)$ to $\mathcal{F}_{transmit}$. If x^k are inconsistent, then $P_{p(id')}$ broadcasts a complaint (bad, id, id').
3. Upon receiving (id'^k_k, x^{*k}) from $\mathcal{F}_{transmit}$, P_k checks if $x^k = x^{*k}$, where x^k is the message that P_k received before the opening. If $x^k \neq x^{*k}$, then P_k broadcasts a complaint (bad, id, id').

If at least one (bad, id, id') is broadcast, or $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{transmit}$, all parties output $(id, id', \perp)$. Otherwise, each party P_k writes $comm^k[id'] \leftarrow comm^k[id]$, and outputs $(\mathsf{confirmed}, id, id')$ to $\mathcal{Z}$. $P_{p(id')}$ outputs (id, id', x) to $\mathcal{Z}$.

Figure 4.18: Real Protocol Π_{commit} (local operations and openings)

Table 4.2: Calls of $\mathcal{F}_{transmit}$ for different functionalities of Π_{commit} with N-bit values

input	called $\mathcal{F}_{transmit}$ functionalities
commit	$\mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
mcommit	$\mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{fwd}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
weak_open	$\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
open	$\mathsf{rev}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
priv_open	$\mathsf{fwd}^{\otimes n}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot N}$
pcommit, lc, trunc	–

At least with the linear (n, t)-threshold schemes used in this thesis (see Section 2.4.2), the overhead of share sizes is multiplicative w.r.t. the bit length of the shared value, i.e. $\mathsf{sh}_n \cdot (M_1 + M_2) = \mathsf{sh}_n \cdot M_1 + \mathsf{sh}_n \cdot M_2$, which means that several values can be shared in parallel without additional overheads to the share size. If $n = 3$, or Shamir's sharing is used, then $\mathsf{sh}_n = 1$.

If we do not use $\mathcal{F}_{transmit}$ as a black box, but look into details of $\Pi_{transmit}$, we see that priv_open can be even cheaper, since the party $p(id')$ that receives all the shares may send just their signatures back to P_k. In this way, private opening can be seen as the second round of the randomness commitment given in Figure 4.9 (and that is how we are going to use priv_open). We have not included the cost of broadcast messages (bad, id) in Table 4.2, since they are sent only in the case when some party attempts to cheat, and their size is insignificant.

Lemma 4.2. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{commit} UC-realizes $\mathcal{F}_{commit}$ in $\mathcal{F}_{transmit}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{commit}$ described in Figure 4.19. The simulator runs a local copy of Π_{commit}, together with a local copy of $\mathcal{F}_{transmit}$. Without loss of generality, let the number of honest parties be $|\mathcal{H}| = t$. If $|\mathcal{H}| > t$, it suffices to take an arbitrary subset of $\mathcal{H}$ of size t. Throughout the simulation, the shares $comm^k[id]$ of $p(id) \in \mathcal{C}$ held by $k \in \mathcal{H}$ should comprise the value $comm[id]$ held by $\mathcal{F}_{commit}$. This ensures that the parties are committed to the values.

Simulatability. All the messages of Π_{commit} are sent through $\mathcal{F}_{transmit}$. Except the accusations (bad, id) and (bad, id, id') that are easy to simulate since $\mathcal{S}$ knows when $\mathcal{A}$ is cheating, all these messages are some shares.

In all commitments, the shares of a corrupted sender are chosen by $\mathcal{A}$. The shares for corrupted receivers $(x^k)_{k \in \mathcal{C}}$ of x belonging to some honest party need to be generated by $\mathcal{S}$ itself. By assumption, Π_{commit} works with a linear (n, t)-threshold sharing scheme with $t = \lceil n/2 \rceil + 1$. Assuming $|\mathcal{C}| < n/2$, there are at most $t - 1$ shares that $\mathcal{S}$ needs to simulate. Since any set of less than t shares looks uniformly distributed, it is sufficient to sample $x^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$.

Let $comm$ be the local array of $\mathcal{F}_{commit}$, and $comm^k$, $k \in [n]$ the local arrays of $\mathcal{S}$ that it stores for each party. Let $\mathcal{H}$ be some fixed set of t honest parties.

• **Initialization and extension**: $\mathcal{S}$ gets (init, m, p) or (ext, m, p) from $\mathcal{F}_{commit}$. Based on these, it initializes its local $\mathcal{F}_{transmit}$.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ should be output for each honest party in Π_{commit}, then $\mathcal{S}$ discards P_k from its local run of Π_{commit}, and assigns $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}, \mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$. In all such cases, it forwards $(\mathsf{cheater}, k)$ to $\mathcal{F}_{commit}$.

• **Public commit:** $\mathcal{S}$ gets (id, x) from $\mathcal{F}_{commit}$. It computes $(x^k)_{k\in[n]} = \mathsf{classify}(x)$ according to the preagreed sharing and writes $comm^k[id] \leftarrow x^k$ for all $k \in [n]$.

• **Commit:** For $p(id) \in \mathcal{C}$, $\mathcal{S}$ gets the shares $(x^k)_{k\in[n]}$ from $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ generates the shares $x^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$ for $k \in \mathcal{C}$. $\mathcal{S}$ simulates distribution of the shares $(x^k)_{k\in[n]}$ using $\mathcal{F}_{transmit}$. If $(\mathsf{cheater}, k)$ should have come from $\mathcal{F}_{transmit}$, $\mathcal{S}$ delivers it to $\mathcal{F}_{commit}$. If no $(\mathsf{cheater}, k)$ has come from $\mathcal{F}_{transmit}$, then all the shares x^k have been successfully delivered. $\mathcal{F}_{commit}$ is waiting for x from $\mathcal{S}$ for $p(id) \in \mathcal{C}$. It may happen that the shares $(x^k)_{k\in[n]}$ coming from $\mathcal{A}$ are inconsistent. $\mathcal{S}$ defines $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$, which is unique since $|\mathcal{H}| = t$. It sends x to $\mathcal{F}_{commit}$ that writes $comm[id] \leftarrow x$. $\mathcal{S}$ writes $comm[id] \leftarrow x$, and $comm^k[id] \leftarrow x^k$ for all $k \in [n]$.

• **Mutual Commit:** If $p(id) \in \mathcal{C}$ [resp. $p(id') \in \mathcal{C}$], then $\mathcal{S}$ gets $(x^k)_{k\in[n]}$ [resp. $(x'^k)_{k\in[n]}$] from $\mathcal{A}$. Otherwise, it generates $x^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$ [resp. $x'^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id')}}$] for all $k \in \mathcal{C}$.
$\mathcal{S}$ handles the obtained shares similarly to the (commit, id, x) case, taking $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$ for $p(id) \in \mathcal{C}$, and $x' \leftarrow \mathsf{declassify}(x'^k)_{k\in\mathcal{H}}$ for $p(id') \in \mathcal{C}$. Messages $(\mathsf{cheater}, k)$ coming from $\mathcal{F}_{transmit}$ are delivered to $\mathcal{F}_{commit}$. If $(id, id', \perp)$ comes from $\mathcal{F}_{commit}$ after the first transmission, it should be $x \neq x'$, and $\mathcal{S}$ simulates broadcasting (bad, id, id').

• **Compute Linear Combination and Truncation:** $\mathcal{S}$ locally performs the computations and assignments for all $k \in \mathcal{C}$. No outputs are produced.

• **Weak Open:** $\mathcal{S}$ gets (id, x) from $\mathcal{F}_{commit}$. For $p(id) \in \mathcal{C}$, all broadcast shares $(x^k)_{k\in[n]}$ are chosen by $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to generate *all* shares $(x^k)_{k\in[n]}$ by itself. Using $comm^k[id]$, and the derivation tree $deriv[id]$, it computes the values x^k for $k \in \mathcal{C}$. It needs to generate the remaining shares x^k for $k \notin \mathcal{C}$ in such a way that $x = \mathsf{declassify}(x^k)_{k\in[n]}$. If $\mathcal{S}$ already has $t-1$ shares of corrupted parties, all the remaining shares are uniquely determined by x and these $t-1$ shares. If it has less than $t-1$ shares, then it needs to generate the missing t' shares by itself. It takes an arbitrary subset $\mathcal{T} \subseteq \mathcal{P} \setminus \mathcal{C}$ of size t'. For all $k \in \mathcal{T}$, it generates $x^k_{id'} \xleftarrow{\$} \mathbb{Z}_{2^{m(id')}}$ for all leaves id' of $deriv[id]$, as it would generate them if it was $k \in \mathcal{C}$, and uses them to compute x^k.
$\mathcal{S}$ simulates $(\mathsf{broadcast}, id^{\mathsf{bc}}_k, x^k)$ using $\mathcal{F}_{transmit}$. If (bad, id) should be broadcast by any party, then $\mathcal{S}$ delivers $(id, \perp)$ to $\mathcal{F}_{commit}$. $\mathcal{S}$ remembers the set of parties $\mathcal{K}$ that have broadcast (bad, id). If $\mathcal{K} \geq t$, then $\mathcal{S}$ delivers $(\mathsf{cheater}, p(id))$ to $\mathcal{F}_{commit}$.

• **Open:** $\mathcal{S}$ gets (id, x) from $\mathcal{F}_{commit}$. If $p(id) \notin \mathcal{C}$, it computes all shares x^k such that $x = \mathsf{declassify}(x^k)_{k\in[n]}$, similarly to $\mathsf{weak\_open}$. For $k \in \mathcal{K}$, $\mathcal{S}$ simulates all the revealings $(\mathsf{reveal}, id^{k'}_k, x^k)$ of the shares $id^{k'}_k$ of leaves of $deriv[id]$; all of those are known to $\mathcal{S}$.

• **Privately Open:** If $p(id') \in \mathcal{C}$, $\mathcal{S}$ gets x from $\mathcal{F}_{commit}$. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ already knows all shares x^k that are stored in $comm[id]$. If $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to simulate these shares. There are up to $t-1$ shares issued to P_k for $k \in \mathcal{C}$, and $\mathcal{S}$ computes the remaining x^k in such a way that $x = \mathsf{declassify}(x^k)_{k\in[n]}$, similarly to $\mathsf{weak\_open}$ and open. For each $k \in \mathcal{C}$, $\mathcal{A}$ chooses the share x^{*k} that should be forwarded by P_k to $P_{p(id')}$. $\mathcal{S}$ simulates all such forwardings using $\mathcal{F}_{transmit}$. If $x^{*k} \neq x^k$ for some k, or $p(id') \in \mathcal{C}$ and $\mathcal{A}$ decides to complain, then the broadcast of (bad, id, id') is simulated. $\mathcal{S}$ delivers $(id, id', \perp)$ to $\mathcal{F}_{commit}$.

Figure 4.19: The simulator $\mathcal{S}_{commit}$

For the messages moving between honest parties, $\mathcal{S}$ only needs to simulate $\mathcal{F}_{transmit}$, which should output the message length to $\mathcal{A}$. The message length can be derived from the bit size $m(id)$ of the ring in which the values are shared.

On inputs weak_open, open (and also priv_open for $p(id') \in \mathcal{C}$), $\mathcal{S}$ needs to simulate to $\mathcal{A}$ all the n shares of some value x. In all these cases, x is given to $\mathcal{S}$ by $\mathcal{F}_{commit}$. For $p(id) \notin \mathcal{C}$, not all shares have been generated yet. If $\mathcal{S}$ already knows all $t-1$ shares x^k of $k \in \mathcal{C}$, then all the other shares can be computed directly from x and these shares. If it has less than $t-1$ shares, then it generates the missing t' shares x^k starting from the leaves id' of $deriv[id]$, as if all $t-1$ corrupted parties were present, just without delivering these shares to $\mathcal{A}$. It does not matter for which honest parties $\mathcal{T}$ exactly such x^k will be generated, all choices are symmetric. Some of the leaf shares $x^k_{id'}$ may already be known to $\mathcal{S}$, if they are coming from public or mutual commitments. All the shares that it does not know have been generated by some honest party, so $x^k_{id'} \xleftarrow{\$} \mathbb{Z}_{2^{m(id')}}$ is a valid distribution for up to $t-1$ shares.

In the case of strong opening (open), $\mathcal{S}$ needs to additionally simulate the shares of leaves of $deriv[id]$ which it does not receive from $\mathcal{F}_{commit}$. Strong opening takes place only if $|\mathcal{K}| \leq t-1$, since otherwise $p(id)$ would be blamed and discarded from $\mathcal{P}$. Hence, all these revealed shares look independent and uniformly distributed to $\mathcal{A}$, and $\mathcal{S}$ may sample $x^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id^{k'}_k)}}$ for the corresponding leaf identifiers $id^{k'}_k$.

Correctness. The delivery of transmitted and broadcast messages is ensured by $\mathcal{F}_{transmit}$. At any time when $(\mathsf{cheater}, k)$ message comes from $\mathcal{F}_{transmit}$, then P_k is discarded from the run of Π_{commit} by all honest parties unanimously. Since $|\mathcal{C}| < n/2$ and $t = \lceil n/2 \rceil + 1$, there is still enough shares to continue running Π_{commit} with the values shared by the other parties. $\mathcal{S}$ sends $(\mathsf{cheater}, k)$ to $\mathcal{F}_{commit}$, so that both the real and the ideal worlds blame P_k and do not perform any computations on its values anymore.

Since all other outputs (not related to cheating) are resulting from some opening, it suffices to show that all opened values are the same in both worlds.

- Let $p(id) \notin \mathcal{C}$. During the opening, at least t shares belonging to $\mathcal{H}$ comprise $comm[id]$. $\mathcal{A}$ may tamper with the shares x^k for $k \in \mathcal{C}$. Since $comm[id]$ is already fixed by the shares of honest parties, $\mathcal{A}$ may at most make the opening inconsistent. In the weak opening case, $\mathcal{A}$ may argue against up to $t-1$ shares of $comm[id]$, but all these complaints will be denied after the strong opening, when the actually transmitted values will be revealed. For $p(id') \notin \mathcal{C}$, if the shares opened during priv_open are consistent (otherwise, $(id, id', \perp)$ is output by both $\mathcal{F}_{commit}$ and Π_{commit}), then the value x reconstructed from these shares equals to the value reconstructed from the shares of $\mathcal{H}$.

- Let $p(id) \in \mathcal{C}$. We need to show that $\mathsf{declassify}(comm^k[id])_{k\in\mathcal{H}} = comm[id]$ for the initially fixed set of honest parties $\mathcal{H}$ is maintained throughout the computation, where $comm^k[id]$ are the shares of the local copy of Π_{commit} of $\mathcal{S}$, and $comm[id]$ is the inner value of $\mathcal{F}_{commit}$. We prove it by induction on the number of operations that have been applied to the shared values.

  - *Base:* The initial values for $comm[id]$ are chosen during executing pcommit, commit, mcommit. For pcommit, x was committed publicly, and $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$ holds by choice of x^k. For commit and mcommit, $\mathcal{S}$ sends to $\mathcal{F}_{commit}$ the value $x \leftarrow \mathsf{declassify}(x^k)_{k\in\mathcal{H}}$, where $x^k = comm^k[id]$ for $k \in \mathcal{H}$ in the local copy of Π_{commit} of $\mathcal{S}$. Hence $\mathsf{declassify}(comm^k[id])_{k\in\mathcal{H}} = comm[id]$.
  - *Step:* The new values $comm[id']$ are created by calling lc and trunc. Since both lc and trunc are linear operations, and we are using linear secret sharing, for $f \in \{\mathsf{lc}, \mathsf{trunc}\}$ we have

    $$\begin{aligned} \mathsf{declassify}(comm^k[id'])_{k\in\mathcal{H}} &= \mathsf{declassify}(f(comm^k[id]))_{k\in\mathcal{H}} \\ &= f(\mathsf{declassify}(comm^k[id]))_{k\in\mathcal{H}} \ . \end{aligned}$$

    By induction hypothesis, $\mathsf{declassify}(comm^k[id]))_{k\in\mathcal{H}} = comm[id]$, and hence this quantity equals $f(comm[id]) = comm[id']$, so we have $\mathsf{declassify}(comm^k[id'])_{k\in\mathcal{H}} = comm[id']$.

  As the result, $\mathcal{A}$ may tamper with the shares x^k for $k \in \mathcal{C}$, but since $comm[id]$ is already fixed by the shares issued to $\mathcal{H}$, they may at most make the opening inconsistent. If it happens in the weak opening case, $\mathcal{S}$ delivers $(id, \perp)$ to $\mathcal{F}_{commit}$, and the opening fails in both worlds. In the strong opening case, $\mathcal{F}_{transmit}$ ensures that only the shares $x^k = comm^k[id]$ that have been indeed received by P_k are opened for $k \notin \mathcal{C}$, and so for $k \in \mathcal{H}$. The case $|\mathcal{K}| \geq t$ may never happen to an honest party, since $|\mathcal{C}| \leq t-1$, and so an honest party will not be blamed even if all of them will cheat. Finally, $comm[id]$ is output to each party in the real protocol. □

4.5.3 Generating Uniformly Distributed Randomness

In our protocols, we need access to some common randomness known to all parties. Moreover, we want this randomness to serve as a challenge for the prover, so that the randomness should be generated in the online phase, *after* the prover makes all its commitments. Hence we try to use the honest majority assumption to make it as efficient as possible. The ideal functionality $\mathcal{F}_{pubrnd}$ for generating public randomness is given in Figure 4.20.

The functionality $\mathcal{F}_{pubrnd}$ works with unique identifiers id, encoding the bit length $m(id)$ of the randomness.

• **Initialization:** On input $(\mathsf{init}, \hat{m})$, assign the mapping $m \leftarrow \hat{m}$. Deliver m to $\mathcal{A}_S$.

• **Randomness commitment:** On input (pubrnd, id) from all (honest) parties, generate a random value $r \in \mathbb{Z}_{2^{m(id)}}$. Output (id, r) to each party, and also to $\mathcal{A}_S$. Alternatively, if $|\mathcal{C}| > 0$, $\mathcal{A}_S$ may choose to output $(\mathsf{cheater}, k)$ for $k \in \mathcal{C}$ to each party instead.

• **Cheater detection:** On input $(\mathsf{cheater}, k)$ from $\mathcal{A}_S$ for $k \in \mathcal{C}$, output $(\mathsf{cheater}, k)$ to all parties. Let $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 4.20: Ideal functionality $\mathcal{F}_{pubrnd}$

The protocol Π_{pubrnd} works with unique identifiers id, encoding the bit length $m(id)$ of the randomness. It uses $\mathcal{F}_{commit}$ as a subroutine.

• **Initialization:** On input $(\mathsf{init}, \hat{m})$, assign the mapping $m \leftarrow \hat{m}$. For all $id \in \mathsf{Dom}(m)$, $j \in [n]$, for $id_j = (id, j)$, assign $m(id_j) \leftarrow m$, $p(id_j) \leftarrow j$. Send (init, m, p) to $\mathcal{F}_{commit}$.

• **Randomness commitment:** On input (pubrnd, id), each party P_i (a set of t parties is sufficient):

1. Generates a random value $r_i \in \mathbb{Z}_{2^{m(id)}}$, and sends $(\mathsf{commit}, id_i, r_i)$ to $\mathcal{F}_{commit}$. For $j \neq i$, it sends (commit, id_j) to $\mathcal{F}_{commit}$.
2. Sends $(id = \sum_{j=1}^{n} id_j)$, and then $(\mathsf{weak\_open}, id)$ to $\mathcal{F}_{commit}$. If $(id, \perp)$ comes back, then it sends (open, id) to $\mathcal{F}_{commit}$.
3. Upon receiving (id, r) from $\mathcal{F}_{commit}$, outputs (id, r) to $\mathcal{Z}$.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{commit}$, each party outputs $(\mathsf{cheater}, k)$ to $\mathcal{Z}$ and discards P_k from the protocol, i.e. $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 4.21: The protocol Π_{pubrnd}

The protocol implementing $\mathcal{F}_{pubrnd}$, built on top of $\mathcal{F}_{commit}$, is given in Figure 4.21. The idea behind the randomness generation is quite standard: each party commits to its own random value as an element of $\mathbb{Z}_{2^m}$, and the sum of these values is opened (alternatively, the parties could generate random bitstrings of certain length and open their bitwise xor).

Lemma 4.3. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{pubrnd} UC-realizes $\mathcal{F}_{pubrnd}$ in $\mathcal{F}_{commit}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{pubrnd}$ described in Figure 4.22. The simulator runs a local copy of Π_{pubrnd} as well as a local copy of $\mathcal{F}_{commit}$.

Simulatability. $\mathcal{S}$ should be able to simulate the randomness r_j of honest parties. By definition, up to the last share, $\mathcal{S}$ samples $r_j \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, so these are distributed uniformly, as $\mathcal{A}$ expects. The last r_j is computed as $r_j = r - \sum_{k=1,k\neq j}^{n} r_k$, where r comes from $\mathcal{F}_{pubrnd}$. Since r is distributed uniformly, so is $r - \sum_{k=1,k\neq j}^{n} r_k$, since even if r_k for $k \in \mathcal{C}$ are not uniform, the value r serves as a mask that makes the final result uniform. At most $t-1$ of values r_j are provided by $\mathcal{A}$, so it is at least one r_j left s.t $j \notin \mathcal{C}$.

• **Initialization:** On input (init, m), $\mathcal{S}$ locally simulates initialization of $\mathcal{F}_{commit}$.

• **Randomness commitment:** On input (pubrnd, id), $\mathcal{S}$ gets (id, r) from $\mathcal{F}_{pubrnd}$. It needs to simulate sending $(\mathsf{commit}, id_j, r_j)$ and (commit, id_j) to $\mathcal{F}_{commit}$. The values r_j for $j \in \mathcal{C}$ are provided by $\mathcal{A}$. The values r_j for $j \notin \mathcal{C}$ need to be simulated by $\mathcal{S}$. Since $\mathcal{A}$ does not expect to see the values r_j generated by $j \notin \mathcal{C}$, $\mathcal{S}$ is free to wait with their generation until $\mathcal{A}$ commits to r_j for all $j \in \mathcal{C}$. After that, $\mathcal{S}$ generates r_j for $j \notin \mathcal{C}$ in such a way that $\sum_{k=1}^{n} r_k = r$. More precisely, up to the last share, it samples $r_j \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, and then computes the last share as $r_j = r - \sum_{k=1, k\neq j}^{n} r_k$. This is always possible if at least t parties contribute r_j.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ should come from $\mathcal{F}_{commit}$, $\mathcal{S}$ delivers $(\mathsf{cheater}, k)$ to $\mathcal{F}_{pubrnd}$. It assigns $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 4.22: The simulator $\mathcal{S}_{pubrnd}$

Correctness. $\mathcal{F}_{pubrnd}$ outputs r to all parties. $\mathcal{S}$ generates r_j of $j \notin \mathcal{C}$ in such a way that $\sum_{k=1}^{n} r_k = r$, so this value is the same in the real and the ideal world. At any time when $(\mathsf{cheater}, k)$ should come from $\mathcal{F}_{commit}$, $\mathcal{S}$ sends the message $(\mathsf{cheater}, k)$ to $\mathcal{F}_{pubrnd}$ that causes honest parties to output $(\mathsf{cheater}, k)$. □

In addition to public randomness, we need to generate randomness that is known only to a certain party (the prover), and that has been stored into $\mathcal{F}_{commit}$. The ideal functionality $\mathcal{F}_{rnd}$ describing it is given in Figure 4.23. Since we want to formally define the impact of $\mathcal{F}_{rnd}$ on $\mathcal{F}_{commit}$, we use $\mathcal{F}_{commit}$ as a *global* resource of the GUC model (see Section 2.2.3), so we denote it $\underline{\mathcal{F}_{commit}}$. GUC allows $\underline{\mathcal{F}_{commit}}$ to be included in both the real and the ideal execution. The protocol Π_{rnd} implementing $\mathcal{F}_{rnd}$ is given in Figure 4.24. It is similar to Π_{pubrnd}, but the shares are opened just to one party.

Differently from $\mathcal{F}_{rnd}$, the protocol Π_{rnd} puts into $\underline{\mathcal{F}_{commit}}$ the intermediate values r_j from which $r = \sum_{j \in [n] \setminus \{p(id)\}} r_j$ is computed. The security proof would clearly fail since r_j are not present in the ideal execution of $\mathcal{F}_{rnd}$ at all. Therefore, we need to constrain the environment $\mathcal{Z}$ to $\mathcal{Z}'$ that will not access the identifiers of $\underline{\mathcal{F}_{commit}}$ that correspond to values generated internally by Π_{rnd}. This is a reasonable assumption, since r_j are needed only for the protocol Π_{rnd}, and the (honest) parties may agree to not use these commitments anywhere else (as it would be in ordinary UC model, if $\mathcal{F}_{commit}$ was a subroutine of Π_{rnd}). There are no restrictions on the identifiers id that are provided by $\mathcal{Z}$ itself. More formally, for all id on which Π_{rnd} is initialized, $\mathcal{Z}'$ it is not allowed to access identifiers of the form id_j of $\underline{\mathcal{F}_{commit}}$ in any way, including computing linear combinations or truncations of such identifiers. When using $\mathcal{F}_{rnd}$ as a subroutine, we ensure that the embedding protocol satisfies this property.

Lemma 4.4. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{rnd} $\underline{\mathcal{F}_{commit}}$-EUC-realizes $\mathcal{F}_{rnd}$, under assumption that $\mathcal{Z}$ does not use any identifiers generated inside Π_{rnd} when accessing $\underline{\mathcal{F}_{commit}}$.*

The functionality $\mathcal{F}_{rnd}$ works with unique identifiers id, encoding the party $p(id)$ committed to the randomness, and the bit length $m(id)$ of the randomness. It stores an array $comm$ of already generated and committed randomness.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$, where $\mathsf{Dom}(\hat{p}) = \mathsf{Dom}(\hat{m})$ assign the mappings $p \leftarrow \hat{p}$, $m \leftarrow \hat{m}$. Deliver (init, m, p) to $\mathcal{A}_S$. Deliver (ext, m, p) to $\underline{\mathcal{F}_{commit}}$.

• **Randomness commitment:** On input (rnd, id) from all (honest) parties, if $comm[id]$ is not defined yet, generate a random value $r \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$. Assign $comm[id] \leftarrow r$. Output (id, r) to $P_{p(id)}$, and $(\mathsf{confirmed}, id)$ to every other party. If $p(id) \in \mathcal{C}$, output (id, r) also to $\mathcal{A}_S$.
Output $\underline{(\mathsf{commit}, id, r)}$ to $P_{p(id)}$, and $\underline{(\mathsf{commit}, id)}$ to each other (honest) party. These messages will be delivered to $\underline{\mathcal{F}_{commit}}$.
Alternatively, if $|\mathcal{C}| > 0$, $\mathcal{A}_S$ may choose to output $(\mathsf{cheater}, k)$ for $k \in \mathcal{C}$ or $(id, \perp)$ to each party.

• **Cheater detection:** On input $(\mathsf{cheater}, k)$ from $\mathcal{A}_S$, output $(\mathsf{cheater}, k)$ to all parties, assign $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$. On input (stop, id) [resp. (stop, id, id')] from $\mathcal{A}_S$, output $(id, \perp)$ [resp. $(id, id', \perp)$] to all parties.

Figure 4.23: Ideal functionality $\mathcal{F}_{rnd}$

The protocol Π_{rnd} works with unique identifiers id, encoding the party $p(id)$ committed to the randomness, and the bit length $m(id)$ of the randomness. It uses a shared instance of $\underline{\mathcal{F}_{commit}}$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$, assign the mappings $m \leftarrow \hat{m}$, $p \leftarrow \hat{p}$. For all $id \in \mathsf{Dom}(m) = \mathsf{Dom}(p)$, for all $j \in [n]$, for $id_j = (id, j)$, assign $\tilde{m}(id_j) \leftarrow m(id)$, $\tilde{p}(id_j) \leftarrow j$, and for $id_j^{'} = (id', j)$, assign $\tilde{m}(id_j^{'}) \leftarrow m(id)$, $\tilde{p}(id_j^{'}) \leftarrow p(id)$. Deliver $(\mathsf{ext}, \tilde{m}, \tilde{p})$ to $\underline{\mathcal{F}_{commit}}$.

• **Randomness commitment:** On input (rnd, id), each party P_i, $i \neq p(id)$ (actually, any fixed set of t parties is sufficient):

1. Generates a random value $r_i \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$ and sends $(\mathsf{commit}, id_i, r_i)$ to $\underline{\mathcal{F}_{commit}}$. For all $i \neq j$, it sends (commit, id_j) to $\underline{\mathcal{F}_{commit}}$.
2. Upon receiving $(\mathsf{confirmed}, id_k)$ from $\underline{\mathcal{F}_{commit}}$ for all $k \neq p(id)$, it sends $(\mathsf{priv\_open}, id_k, id_k^{'})$ for all $k \neq p(id)$ to $\underline{\mathcal{F}_{commit}}$.
3. Upon receiving $(\mathsf{confirmed}, id_k, id_{k'})$ for all k, k', it sends $(id = \sum_{k=1, k\neq p(id)}^{n} id_k^{'})$ to $\underline{\mathcal{F}_{commit}}$.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$, $(id, \perp)$, or $(id, id', \perp)$ comes from $\underline{\mathcal{F}_{commit}}$, each party outputs it to $\mathcal{Z}$, and discards P_k from the protocol run, i.e. assigns $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 4.24: The protocol Π_{rnd}

• **Initialization:** On input (init, m, p), $\mathcal{S}$ simulates initialization of a new session of $\underline{\mathcal{F}_{commit}}$.

• **Randomness commitment:** On input (rnd, id), $\mathcal{S}$ needs to simulate sending $(\mathsf{commit}, id_j, r_j)$ and $(\mathsf{priv\_open}, id_j, id_j^{'})$ to $\underline{\mathcal{F}_{commit}}$. The values r_j for $j \in \mathcal{C}$ are provided by $\mathcal{A}$, and r_j for $j \notin \mathcal{C}$ do not have to be opened to $\mathcal{A}$ yet. After $\mathcal{A}$ has provided r_j for all $j \in \mathcal{C}$, and $(\mathsf{confirmed}, id_j)$ has been output to all parties, $\mathcal{A}$ waits for $(\mathsf{priv\_open}, id_j, id_j^{'})$ for all $j \in [n] \setminus \{p(id)\}$. Since $\mathcal{S}$ has already received all r_j for $j \in \mathcal{C}$, it makes r_j for $j \notin \mathcal{C}$ dependent on r_j for $j \in \mathcal{C}$, exactly in the same way as it was done by $\mathcal{S}_{pubrnd}$.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ should come from $\underline{\mathcal{F}_{commit}}$, $\mathcal{S}$ delivers $(\mathsf{cheater}, k)$ to $\mathcal{F}_{rnd}$, and discards P_k from Π_{rnd}, i.e. $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 4.25: The simulator $\mathcal{S}_{rnd}$

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{rnd}$ described in Figure 4.25. The simulator runs a local copy of Π_{rnd}, and it has a limited access to the global $\underline{\mathcal{F}_{commit}}$ (via the adversarial ports). It also runs its own local copy of $\mathcal{F}_{commit}$, trying to convince $\mathcal{A}$ that all messages coming from it originate from $\underline{\mathcal{F}_{commit}}$.

Simulatability. For $p(id) \in \mathcal{C}$, $\mathcal{S}$ gets r from $\mathcal{F}_{rnd}$. $\mathcal{S}$ should be able to simulate the randomness r_j generated by honest parties in such a way that r will be finally output to $P_{p(id)}$ in the simulated Π_{rnd}. In this case, the generation of appropriate r_j is done exactly in the same settings as for $\mathcal{S}_{pubrnd}$, so we refer to the proof of Lemma 4.3 here. For $p(id) \notin \mathcal{C}$, the value r does not have to be simulated to $\mathcal{A}$ anyway.

All the commitments of r_j are simulated using the *local* copy of $\mathcal{F}_{commit}$. Differently from $\mathcal{S}_{pubrnd}$, we now have a global $\underline{\mathcal{F}_{commit}}$ in the ideal world, and the environment may send any inputs to it, including (open, id). Since we agreed that $\mathcal{Z}'$ is not allowed to access the internal identifiers of Π_{rnd} in any way, it cannot make any queries involving r_j, and hence distinguish the outputs given to $\mathcal{A}$ by the simulated $\mathcal{F}_{commit}$ from the outputs that $\mathcal{A}$ would get from $\underline{\mathcal{F}_{commit}}$. It is only allowed to open the final randomness r.

Correctness. $\mathcal{F}_{rnd}$ outputs (id, r) to $P_{p(id)}$. In addition, it outputs a message $\underline{(\mathsf{commit}, id, r)}$ to $P_{p(id)}$, and $\underline{(\mathsf{commit}, id)}$ to all other (honest) parties. These messages are delivered to $\underline{\mathcal{F}_{commit}}$ that writes $comm[id] = r$. In the EUC model, $\mathcal{Z}$ may send (open, id) to $\underline{\mathcal{F}_{commit}}$ and check whether r is the same as $\mathcal{A}$ has reported. $\mathcal{S}$ generates r_j of $j \notin \mathcal{C}$ in such a way that $\sum_{k=1,k\neq p(id)}^{n} r_k = r$, so the values output to $P_{p(id)}$ are the same in the simulation, and $\mathcal{A}$ believes that it should be $comm[id] = r$ in the inner state of $\mathcal{F}_{commit}$ in the real protocol.

At any time when $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{commit}$, $\mathcal{S}$ sends $(\mathsf{cheater}, k)$ to $\mathcal{F}_{rnd}$ that causes honest parties to output $(\mathsf{cheater}, k)$. If priv_open fails, then outputting $(id, id', \perp)$ is simulated in Π_{rnd}, and $\mathcal{S}$ sends (stop, id, id') to $\mathcal{F}_{rnd}$, so that $(id, id'\perp)$ is output also in the ideal world. Hence all outputs are the same in $\mathcal{F}_{rnd}$ and Π_{rnd}. □

Observation 4.3. From the protocols Π_{pubrnd} and Π_{rnd}, one can read out the numbers of $\mathcal{F}_{commit}$ operations needed for generating N-bit randomness. We can use the results of Table 4.2 to translate them directly to $\mathcal{F}_{transmit}$ operations. Let comm_M, wopen_M open_M, popen_M denote the calls of commit, weak_open, open, priv_open respectively on an M-bit value. The results are given in Table 4.3.

4.5.4 Generation of Precomputed Tuples

The 3-party version of the functionality $\mathcal{F}_{pre}$ used to generate and commit a sufficient amount of verified precomputed multiplication triples and trusted bits has been given in Figure 4.5. Figure 4.26 depicts its generalized n-party version.

Table 4.3: Calls of $\mathcal{F}_{commit}$ and $\mathcal{F}_{transmit}$ for different functionalities of Π_{pubrnd} and Π_{rnd} for generating N-bit randomness

input	$\mathcal{F}_{commit}$ calls	$\mathcal{F}_{transmit}$ calls
rnd	$\mathsf{comm}_N^{\otimes t} \oplus \mathsf{popen}_N^{\otimes t}$	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nt} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot N}^{\otimes nt} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nt}$
pubrnd (cheap)	$\mathsf{comm}_N^{\otimes t} \oplus \mathsf{wopen}_N$	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nt} \oplus \mathsf{bc}_{\mathsf{sh}_n \cdot N}^{\otimes n}$
pubrnd (expensive)	$\mathsf{comm}_N^{\otimes t} \oplus \mathsf{open}_N$	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes nt} \oplus \mathsf{rev}_{\mathsf{sh}_n \cdot N}^{\otimes n}$

Table 4.4: Number of tuple bits involved in different steps (ring cardinality 2^m)

x	$\mathsf{nb}_{\mathsf{tpl}}(x,m)$	$\mathsf{nb}_{\mathsf{op}_1}(x,m)$	$\mathsf{nb}_{\mathsf{op}_2}(x,m)$
bit	m	1	m
triple	$3m$	$2m$	m

Similarly to $\mathcal{F}_{rnd}$ of Section 4.5.3, we use a shared functionality $\underline{\mathcal{F}_{commit}}$ as a global resource, to make it possible to use the generated tuples later. The protocol Π_{pre} implementing $\mathcal{F}_{pre}$ is formalized in Figure 4.27-4.28. A single identifier id corresponds to generating $u(id)$ tuples in a ring $\mathbb{Z}_{m(id)}$ for the prover $P_{p(id)}$. The party $P_{p(id)}$ first generates $\mu \cdot u(id) + \kappa$ tuples itself, and only $u(id)$ of these tuples are left after the cut-and-choose and the pairwise verification.

Similarly to Π_{rnd} of Section 4.5.3, if we do not put any constraints on $\mathcal{Z}$, this protocol is clearly insecure in $\underline{\mathcal{F}_{commit}}$-EUC model since Π_{pre} generates a lot of intermediate tuples that are used for cut-and-choose and pairwise verification only. If $\mathcal{Z}$ was allowed to open them, it would distinguish Π_{pre} from $\mathcal{F}_{pre}$. Intuitively, these discarded tuples cannot be reused in any outer protocol anyway, since they have already been used as masks, and reusing them would break privacy of the accepted tuples. Hence, similarly to Π_{rnd}, we again put a constraint on $\mathcal{Z}$, allowing it to access only the $u(id)$ finally accepted tuples. When using $\mathcal{F}_{pre}$ as subroutine, we ensure that the embedding protocol satisfies this property.

Observation 4.4. From the description of Π_{pre}, we can extract the total number $\mathsf{nb}_{\mathsf{tpl}}(T)$ of bits needed to encode a single tuple of type T, and the numbers $\mathsf{nb}_{\mathsf{op}_1}(T)$ and $\mathsf{nb}_{\mathsf{op}_2}(T)$ of bits to be opened in the pairwise check, where $\mathsf{nb}_{\mathsf{op}_1}(T)$ bits are opened before the last $\mathsf{nb}_{\mathsf{op}_2}(T)$ bits. These values are given in Table 4.4.

Lemma 4.5 (cost of precomputed tuple generation of Π_{pre})**.** *Let $\underline{\mathcal{F}_{commit}}$ be realized by Π_{commit}. Let λ be the number of bits in the randomness seed. Given the parameters μ and κ, the number of $\mathcal{F}_{transmit}$ operations for generating N tuples of type T using Π_{pre} can be expressed as the quantity prc_T^N defined as*

$$\begin{aligned}\mathsf{prc}_T^N \quad = \quad & \mathsf{tr}_{(\mu N+\kappa)\cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{tpl}} T)}^{\otimes n} \otimes (\mathsf{tr}_{\mathsf{sh}_n \cdot \lambda}^{\otimes nt} \oplus \mathsf{bc}_{\mathsf{sh}_n \cdot \lambda}^{\otimes n}) \\ & \oplus (\mathsf{bc}_{\kappa \cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{tpl}} T)}^{\otimes n} \otimes \mathsf{bc}_{(\mu-1) N \cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{op}_1} T)}^{\otimes n} \otimes \mathsf{bc}_{(\mu-1) N \cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{op}_2} T)}^{\otimes n}) \ .\end{aligned}$$

$\mathcal{F}_{pre}$ works with unique identifiers id, encoding a bit size $m(id)$ of the ring in which the tuples are committed, the party $p(id)$ that gets all the tuples, and the number $u(id)$ of tuples to be generated. It stores an array $comm$ of already generated tuples.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{u}, \hat{p})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{u}) = \mathsf{Dom}(\hat{p})$, assign the mappings $m \leftarrow \hat{m}$, $u \leftarrow \hat{u}$, $p \leftarrow \hat{p}$. Deliver $\hat{m}$, $\hat{u}$, $\hat{p}$ to $\mathcal{A}_S$. For each id, define a couple of identifiers id_i^k for $k \in [u(id)]$, and $i \in [v]$, where $v = 1$ for trusted bits, and $v = 3$ for triples. Define $\tilde{m}(id_i^k) \leftarrow m(id)$, $\tilde{p}(id_i^k) \leftarrow p(id)$ for all i,k. Send $(\mathsf{ext}, \tilde{m}, \tilde{p})$ to $\underline{\mathcal{F}_{commit}}$.

• **Trusted bits:** On input (bit, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate a vector of random bits $\vec{b} \xleftarrow{\$} \mathbb{Z}_2^{u(id)}$. If $p(id) \in \mathcal{C}$, get $\vec{b} \in \mathbb{Z}_2^{u(id)}$ from $\mathcal{A}_S$.
2. Assign $comm[id] \leftarrow \vec{b}$.
3. Output $\vec{b}$ to $P_{p(id)}$. If $p(id) \in \mathcal{C}$, output $\vec{b}$ also to $\mathcal{A}_S$.
4. For all $k \in [u(id)]$, output $\underline{(\mathsf{commit}, id_0^k, b_k)}$ to $P_{p(id)}$, and $\underline{(\mathsf{commit}, id_0^k)}$ to each other (honest) party. These messages will be delivered to $\underline{\mathcal{F}_{commit}}$.

• **Multiplication triples:** On input (triple, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate $\vec{a} \xleftarrow{\$} \mathbb{Z}_{m(id)}^{u(id)}$, $\vec{b} \xleftarrow{\$} \mathbb{Z}_{m(id)}^{u(id)}$. If $p(id) \in \mathcal{C}$, get $\vec{a}$ and $\vec{b}$ from $\mathcal{A}_S$. Compute elementwise $\vec{c} = \vec{a} \cdot \vec{b}$.
2. Assign $comm[id] = (\vec{a}, \vec{b}, \vec{c})$.
3. Output $(\vec{a}, \vec{b}, \vec{c})$ to $P_{p(id)}$. If $p(id) \in \mathcal{C}$, output $(\vec{a}, \vec{b}, \vec{c})$ also to $\mathcal{A}_S$.
4. For all $k \in [u(id)]$, output $\underline{(\mathsf{commit}, id_0^k, a_k)}$, $\underline{(\mathsf{commit}, id_1^k, b_k)}$, $\underline{(\mathsf{commit}, id_2^k, c_k)}$ to $P_{p(id)}$, and $\underline{(\mathsf{commit}, id_0^k)}$, $\underline{(\mathsf{commit}, id_1^k)}$, $\underline{(\mathsf{commit}, id_2^k)}$ to each other (honest) party. These messages will be delivered to $\underline{\mathcal{F}_{commit}}$.

• **Stopping:** On input (stop, id) from $\mathcal{A}_S$, stop the functionality and output $(id, \perp)$ to all parties.

Figure 4.26: Ideal functionality $\mathcal{F}_{pre}$

In Π_{pre}, each party works with unique identifiers id, encoding the bit size $m(id)$ of the ring in which the tuples are committed, the party $p(id)$ that gets all the tuples, and the number $u(id)$ of tuples to be generated. Π_{pre} uses $\mathcal{F}_{pubrnd}$ as a subroutine, and $\underline{\mathcal{F}_{commit}}$ as a shared subroutine. The parameters μ and κ depend on the security parameter. Let λ be the number of bits in the randomness generator seed.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{u}, \hat{p})$ from $\mathcal{Z}$, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{u}) = \mathsf{Dom}(\hat{p})$, each party assigns the functions $m \leftarrow \hat{m}$, $u \leftarrow \hat{u}$, $p \leftarrow \hat{p}$. For each id, it defines a couple of identifiers id_i^k for $k \in [\mu \cdot u(id) + \kappa]$, and $i \in [v]$, where $v = 1$ for trusted bits, and $v = 3$ for triples. It defines $\tilde{m}(id_i^k) \leftarrow m(id)$, $\tilde{p}(id_i^k) \leftarrow p(id)$ for all i,k. It sends $(\mathsf{ext}, \tilde{m}, \tilde{p})$ to $\underline{\mathcal{F}_{commit}}$.
In addition, each party defines $\tilde{m}(\mathsf{k}) = 2^\lambda$ for some constant identifier k, and sends $(\mathsf{init}, \tilde{m})$ to $\mathcal{F}_{pubrnd}$.

• **Stopping:** If at any time $(\mathsf{cheater}, k)$ comes from $\underline{\mathcal{F}_{commit}}$ or $\mathcal{F}_{pubrnd}$ while executing (bit, id) or (triple, id), output $(id, \perp)$ to $\mathcal{Z}$.

Figure 4.27: Real protocol Π_{pre} (initialization and stopping)

• **Trusted bits:** On input (bit, id):

1. The party $P_{p(id)}$ generates $(\mu \cdot u(id) + \kappa)$ random bits $b_k \xleftarrow{\$} \mathbb{Z}_2$. $P_{p(id)}$ sends $(\mathsf{commit}, id_0^k, b_k)$ to $\underline{\mathcal{F}_{commit}}$. Each party sends $(\mathsf{commit}, id_0^k)$ to $\underline{\mathcal{F}_{commit}}$. $P_{p(id)}$ writes $comm[id_0^k] \leftarrow b_k$.
2. The parties send $(\mathsf{pubrnd}, \mathsf{k})$ to $\mathcal{F}_{pubrnd}$, getting back a randomness seed that they use to agree on a random permutation π of tuple indices.
3. For $k \in [\kappa]$, each party sends $(\mathsf{weak\_open}, id_0^{\pi_k})$ to $\underline{\mathcal{F}_{commit}}$, getting back a bit b_k. If the opening fails, or $b_k \notin \{0, 1\}$, then output $(id, \perp)$.
4. Taking the next $2 \cdot u(id)$ entries of π, the parties partition the corresponding bits into pairs. Such pairwise verification is repeated $\mu - 1$ times with the same $u(id)$ bits, each time taking the next $u(id)$ indices from π.

   For each pair (k, k'), $P_{p(id)}$ broadcasts a bit indicating whether $b_k = b_{k'}$ or not. If $b_k = b_{k'}$ was indicated, each party sends $(id_0^{k,k'} = id_0^k - id_0^{k'})$ to $\underline{\mathcal{F}_{commit}}$. If $b_k \neq b_{k'}$ was indicated, each party sends $(id_0^{k,k'} = 1 - id_0^k - id_0^{k'})$ to $\underline{\mathcal{F}_{commit}}$. Each party then sends $(\mathsf{weak\_open}, id_0^{k,k'})$ to $\underline{\mathcal{F}_{commit}}$ and checks if the value returned by $\underline{\mathcal{F}_{commit}}$ equals 0. If it does not, then output $(id, \perp)$.
5. Let $\vec{id}$ be the vector of the identifiers of the remaining $u(id)$ bits in $\underline{\mathcal{F}_{commit}}$. For $id' \in \vec{id}$, $P_{p(id)}$ outputs $comm[id']$.

• **Multiplication triples:** On input (triple, id):

1. The party $P_{p(id)}$ generates $(\mu \cdot u(id) + \kappa)$ random ring element pairs $a_k \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, $b_k \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$. It computes $c_k = a_k \cdot b_k$ for $k \in |\mu \cdot u(id) + \kappa|$. $P_{p(id)}$ sends $(\mathsf{commit}, id_0^k, a_k)$, $(\mathsf{commit}, id_1^k, b_k)$, $(\mathsf{commit}, id_2^k, c_k)$ to $\underline{\mathcal{F}_{commit}}$. Each party sends $(\mathsf{commit}, id_0^k)$, $(\mathsf{commit}, id_1^k)$, $(\mathsf{commit}, id_2^k)$ to $\underline{\mathcal{F}_{commit}}$. $P_{p(id)}$ writes $comm[id_0^k] \leftarrow a_k$, $comm[id_1^k] \leftarrow b_k$, $comm[id_2^k] \leftarrow c_k$.
2. The parties send $(\mathsf{pubrnd}, \mathsf{k})$ to $\mathcal{F}_{pubrnd}$, getting back a randomness seed that they use to agree on a random permutation π of tuple indices.
3. For $k \in [\kappa]$, each party sends $(\mathsf{weak\_open}, id_0^{\pi_k})$, $(\mathsf{weak\_open}, id_1^{\pi_k})$, $(\mathsf{weak\_open}, id_2^{\pi_k})$ to $\underline{\mathcal{F}_{commit}}$, getting back (a_k, b_k, c_k). If the opening fails, or $c_k \neq a_k \cdot b_k$, then output $(id, \perp)$.
4. Taking the next $2 \cdot u(id)$ entries of π, the parties partition the corresponding triples into pairs. Such pairwise verification is repeated $\mu - 1$ times with the same $u(id)$ triples, each time taking the next $u(id)$ indices from π.

   For each pair (k, k'), let us denote $(id^a, id^b, id^c) = (id_0^k, id_1^k, id_2^k)$, $(id^{a'}, id^{b'}, id^{c'}) = (id_0^{k'}, id_1^{k'}, id_2^{k'})$, $(id^{\hat{a}}, id^{\hat{b}}, id^{\hat{c}}) = (id_0^{k,k'}, id_1^{k,k'}, id_2^{k,k'})$.

   (a) Each party sends $(id^{\hat{a}} = id^a - id^{a'})$, $(id^{\hat{b}} = id^b - id^{b'})$, and then $(\mathsf{weak\_open}, id^{\hat{a}})$, $(\mathsf{weak\_open}, id^{\hat{b}})$ to $\underline{\mathcal{F}_{commit}}$, getting back $\hat{a}$ and $\hat{b}$ respectively.

   (b) Each party then sends $(id^{\hat{c}} = \hat{a} \cdot id^b + \hat{b} \cdot id^{a'} + id^{c'} - id^c)$ and $(\mathsf{weak\_open}, id^{\hat{c}})$ to $\underline{\mathcal{F}_{commit}}$. If $\underline{\mathcal{F}_{commit}}$ returns $\hat{c} \neq 0$, output $(id, \perp)$.
5. Let $\vec{id}$ be the vector of the identifiers of the remaining $u(id)$ triples in $\underline{\mathcal{F}_{commit}}$. For $id' \in \vec{id}$, $P_{p(id)}$ outputs $comm[id']$.

Figure 4.28: Real protocol Π_{pre} (tuple generation)

Proof. From Table 4.2, we get the cost $\mathsf{tr}^{\otimes n}_{\mathsf{sh}_n \cdot M}$ of sharing an M-bit value using $\mathcal{F}_{commit}$, and the cost $\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot M}$ of weak opening an M-bit value using $\mathcal{F}_{commit}$. The lc operations do not involve any communication. The sum prc^N_T covers all the communication for generating all the N triples of type T.

- $\mathsf{tr}^{\otimes n}_{(\mu N+\kappa)\cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{tpl}} T)}$ is the cost of sharing the initial unverified tuples among the n parties in parallel.
- $\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot \lambda} \oplus \mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot \lambda}$ is the cost of agreeing on a λ-bit common randomness seed using weak opening.
- $\mathsf{bc}^{\otimes n}_{\kappa \cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{tpl}} T)}$ is the cost of cut-and-choose weak opening. All the κ tuples are opened in parallel.
- $\mathsf{bc}^{\otimes n}_{(\mu-1)N \cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{op}_1} T)}$ and $\mathsf{bc}^{\otimes n}_{(\mu-1)N \cdot \mathsf{sh}_n \cdot (\mathsf{nb}_{\mathsf{op}_2} T)}$ are the costs of the pairwise verifications of all the $(\mu - 1)$ pairs, which counts the total cost of the two weak openings of this step: the differences between the two tuples, and the alleged zeroes. For trusted bits, the share cost multiplier sh_n can be removed from $\mathsf{nb}_{\mathsf{op}_1} T$ since the difference between two bits is broadcast in plain, not as shares.

Since agreeing on public randomness using Π_{pubrnd} takes more than one round, and the randomness is not opened to any party in the first round, the steps (1) and (2) of Π_{pre} can be done in parallel. Since all communication of $\mathsf{weak\_open}$ in Π_{commit} originates from the prover, and computing linear combinations using $\mathcal{F}_{commit}$ does not introduce any communication, the steps (3) and (4) of Π_{pre} can also be done in parallel. □

Lemma 4.6. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, if $\mu > 1 + \eta/\log N$ and $\kappa > \max\{(N^{1/\mu} + 1)\eta, N^{1/\mu} + \mu - 1\}$, where N is the total number of generated tuples, the protocol Π_{pre} $\mathcal{F}_{commit}$-EUC-realizes $\mathcal{F}_{pre}$ in $\mathcal{F}_{pubrnd}$-hybrid model with correctness error $\varepsilon < 2^{\eta}$, and simulation error 0.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{pre}$ described in Figure 4.29. The simulator runs a local copy of Π_{pre}, together with local copies of $\mathcal{F}_{pubrnd}$ and $\mathcal{F}_{commit}$.

Simulatability. The simulator will need to generate some non-trivial values during the openings of the cut-and-choose and the pairwise verification, so it should be prepared. During the initial distribution of tuples, for $p(id) \notin \mathcal{C}$, it generates $(\mu - 1)u(id) + \kappa$ additional valid tuples. $\mathcal{S}$ generates a random seed s, computes the permutation π from s, and rearranges the tuples in such a way that exactly those tuples that are chosen by $\mathcal{F}_{pre}$ will be finally left. For $p(id) \in \mathcal{C}$, all $\mu \cdot u(id) + \kappa$ tuples are chosen by $\mathcal{A}$. After $\mathcal{S}$ gets all these tuples from $\mathcal{A}$, it

• **Initialization:** On input (init, m, u, p) from $\mathcal{F}_{pre}$, $\mathcal{S}$ initializes a session of an internal $\mathcal{F}_{commit}$.

• **Tuple generation:** On input (bit, id) and (triple, id), $\mathcal{S}$ behaves according to the following pattern:

1. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ generates $(\mu - 1)u(id) + \kappa$ additional tuples. It then generates a random seed $s \xleftarrow{\$} \mathbb{Z}_{2^\lambda}$, and checks which permutation π is generated by s. It then permutes all $\mu \cdot u(id) + \kappa$ tuples (some $u(id)$ of them are "imaginary" tuples that are not known to $\mathcal{S}$) in such a way that the $u(id)$ tuples that will be finally left (based on π) are exactly those generated by $\mathcal{F}_{pre}$. It simulates committing them to $\underline{\mathcal{F}_{commit}}$.
   If $p(id) \in \mathcal{C}$, then all the $\mu u(id) + \kappa$ tuples are chosen by $\mathcal{A}$.
2. The parties should jointly agree on a random permutation π of tuples.
   - In order to agree on the same π, $\mathcal{S}$ simulates $\mathcal{F}_{pubrnd}$ in such a way that it provides the same randomness seed s that $\mathcal{S}$ used to rearrange the tuples before the commitments.
   - Now a vector $\vec{s}'$ of certain κ tuples needs to be revealed. $\mathcal{S}$ needs to simulate the weak opening of $\mathcal{F}_{commit}$, but that requires knowing the values $\vec{s}'$ to be opened. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ takes the $\vec{s}'$ chosen by $\mathcal{A}$ before. If $p(id) \notin \mathcal{C}$, then $\mathcal{S}$ simulates opening the random valid tuples whose commitment it has simulated before. $\mathcal{S}$ either accepts or rejects the opened tuples from the name of honest parties, exactly in the same way as they would do in Π_{pre}. If any tuple should be rejected, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{pre}$.
3. The parties start verifying the tuples pairwise. For this, certain values should be computed and opened using $\mathcal{F}_{commit}$, that depend on the tuple type. For $p(id) \in \mathcal{C}$, $\mathcal{S}$ already knows these values, and hence can simulate their opening. If there are any inconsistencies, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{pre}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ needs to simulate two types of values:
   - The first component are alleged zeroes. For these, $\mathcal{S}$ simulates opening 0.
   - The second component are some additional values needed in verification. For these, $\mathcal{S}$ simulates opening uniformly distributed random values in the corresponding rings.
4. There are now $u(id)$ tuples left for each party that are treated as the final output. For $p(id) \in \mathcal{C}$, $\mathcal{F}_{pre}$ outputs to $P_{p(id)}$ and $\mathcal{S}$ a vector $\vec{s}$ of valid tuples.

• **Stopping:** If at any time $(\mathsf{cheater}, k)$ comes from $\underline{\mathcal{F}_{commit}}$ or $\mathcal{F}_{pubrnd}$, output (stop) to $\mathcal{F}_{pre}$.

Figure 4.29: The simulator $\mathcal{S}_{pre}$

simulates the commitments using $\underline{\mathcal{F}_{commit}}$. For this, it does not need to know the values of the tuples of $p(id) \notin \mathcal{C}$ produced by $\mathcal{F}_{pre}$. The work proceeds as follows.

1. $\mathcal{S}$ initializes $\mathcal{F}_{pubrnd}$ and simulates its run in such a way that the seed s will be the same that was used by $\mathcal{S}$ in the beginning. Since $\mathcal{S}$ has generated s uniformly, this is what $\mathcal{A}$ expects to get from $\mathcal{F}_{pubrnd}$.
2. For cut-and-choose of honest provers, $\mathcal{S}$ generates the opened tuples from the same distribution as an honest prover would. By choice of the randomness seed s, these tuples are completely new and are not related to the $u(id)$ tuples generated by $\mathcal{F}_{pre}$.

3. For the pairwise verification, $\mathcal{S}$ needs to simulate different values, depending on the tuple type. For the first $\mu - 1$ iterations, $\mathcal{S}$ generates all the tuples for honest parties, since they are not included into $\mathcal{F}_{pre}$ anyway. The most interesting is the last μ-th iteration. Let k be the tuple that will be finally output and is not known to $\mathcal{S}$, and let k' be the tuple that $\mathcal{S}$ may still choose itself.

   (a) *Trusted bits:* First of all, $\mathcal{S}$ needs to broadcast a bit indicating whether $b_k \neq b_{k'}$. This value is distributed uniformly since $b_{k'}$ has not been used anywhere yet. After that, $\mathcal{S}$ simulates $\mathcal{F}_{commit}$ outputting either $b_k - b_{k'}$, or $1 - b_k - b_{k'}$. For an honest prover, that value is always 0 since it tells honestly if $b_k \neq b_{k'}$.

   (b) *Multiplication triples:* $\mathcal{S}$ broadcasts $\hat{a} = a_k - a_{k'}$ and $\hat{b} = b_k - b_{k'}$ which are uniformly distributed due to the masks $a_{k'}$ and $b_{k'}$. For an honest prover, the value $\hat{a} \cdot b_k + \hat{b} \cdot a_{k'} + c_{k'} - c_k$ equals 0, since it would generate $c_k = a_k \cdot b_k$ and $c_{k'} = a_{k'} \cdot b_{k'}$.

   Similarly to $\mathcal{S}_{rnd}$, all these simulations would not work if $\mathcal{Z}$ had access to the inner identifiers of Π_{pre} and could open all the intermediate tuples that were used as masks. Since we have allowed $\mathcal{Z}$ to access only the outer identifiers, which correspond to the finally accepted tuples that $\mathcal{Z}$ would get from the party outputs anyway, there is no additional information that it may extract from $\underline{\mathcal{F}_{commit}}$.

Correctness. One difference between $\mathcal{F}_{pre}$ and Π_{pre} is that $\mathcal{F}_{pre}$ outputs $\underline{(\mathsf{commit}, id, x)}$ and $\underline{(\mathsf{commit}, id)}$ only if $(id, \perp)$ is not output to all parties. On the contrary, Π_{pre} commits all tuples already in the beginning, and the protocol just checks whether these tuples are correct. Since we have already agreed that $\mathcal{Z}$ does not access the temporary triples, we may treat all unconfirmed tuples as temporary, and just copy them to the "external" identifier id after they have been accepted. This is just a formality, and not a real problem.

If the protocol succeeds for $p(id) \notin \mathcal{C}$, the finally left $u(id)$ tuples are exactly those that are generated by $\mathcal{F}_{pre}$, so the outputs are the same in both worlds. For $p(id) \in \mathcal{C}$, these $u(id)$ tuples are all generated by $\mathcal{A}$. If any of these tuples is invalid, there will be immediate difference with $\mathcal{F}_{pre}$ that outputs $\underline{(\mathsf{commit}, id, x)}$ and $\underline{(\mathsf{commit}, id)}$ to $\underline{\mathcal{F}_{commit}}$ only for those x that correspond to valid tuples. We need to show that, if finally $u(id)$ tuples are accepted for $p(id) \in \mathcal{C}$, then they are all valid, except with negligible probability.

First of all, we show that, if the tuple with the index k' is valid, then the pairwise check passes only if the tuple k is also valid. We prove it for different kinds of tuples, one by one.

1. *Trusted bits:* Let $b_{k'} \in \{0,1\}$. First, the prover decides whether to indicate $b_k = b_{k'}$, or $b_k \neq b_{k'}$.

   - Suppose the prover indicated $b_k = b_{k'}$. In this case, $b_k - b_{k'}$ is output. If indeed $b_k - b_{k'} = 0$, then it should be $b_k = b_{k'} \in \{0,1\}$.
   - Suppose the prover indicated $b_k \neq b_{k'}$. In this case, $1 - b_k - b_{k'}$ is output. If indeed $1 - b_k - b_{k'} = 0$, then $b_k = 1 - b_{k'} \in \{0,1\}$.
   - If the prover indicates something else, the protocol aborts. No tuples are accepted.

2. *Multiplication triples:* Let $c_{k'} = a_{k'} \cdot b_{k'}$. The values $\hat{a} = a_k - a_{k'}$ and $\hat{b} = b_k - b_{k'}$ are computed and opened by the parties using $\mathcal{F}_{commit}$, so there is no way to cheat with them. Suppose that $\hat{a} \cdot b_k + \hat{b} \cdot a_{k'} + c_{k'} - c_k = 0$. Since $c_{k'} = a_{k'} \cdot b_{k'}$, we have $\hat{a} \cdot b_k + \hat{b} \cdot a_{k'} + a_{k'} \cdot b_{k'} - c_k = (a_k - a_{k'}) \cdot b_k + (b_k - b_{k'}) \cdot a_{k'} + a_{k'} \cdot b_{k'} - c_k = a_k \cdot b_k - c_k$. If this value is 0, then $a_k \cdot b_k = c_k$.

We have shown that the only possibility for the prover to cheat is to put two invalid tuples into the same pair during the pairwise verification. For the $\mu - 1$ pairwise checks, the finally accepted invalid tuple should be paired with some other invalid tuple on each iteration. Now we need to show that, for sufficiently large μ and κ, this happens only with a negligible probability.

Let $p(id) \in \mathcal{C}$. Let $u := u(id)$. If $P_{p(id)}$ wants to have ℓ bad tuples among the final u ones, it has to do the following:

1. put $\mu \cdot \ell$ bad tuples into the initial set of $(\mu \cdot u + \kappa)$ tuples;
2. hope that no bad tuple is among the ones opened during the cut-and-choose step;
3. hope that the $\mu \cdot \ell$ tuples are put together into ℓ groups during the pairwise checking step.

We now give lower bounds for the values μ and κ, such that, from the point of view of a malicious prover, the probability of steps (2) and (3) succeeding is less than $2^{-\eta}$ for a security parameter η.

Probability of passing cut-and-choose. The κ tuples to be opened can be selected in $\binom{\mu u + \kappa}{\kappa}$ different ways. Assuming that some ℓ of the u tuples are "bad", there are $\binom{\mu(u-\ell)+\kappa}{\kappa}$ ways of choosing a set that contains only "good" tuples.

The probability of selecting such a set is

$$\begin{aligned} P_{\text{c\&c}}(\mu,u,\kappa,\ell) &= \frac{\binom{\mu(u-\ell)+\kappa}{\kappa}}{\binom{\mu u+\kappa}{\kappa}} \\ &= \frac{(\mu(u-\ell)+\kappa)!}{(\mu u+\kappa)!}\cdot\frac{(\mu u)!}{(\mu(u-\ell))!} \\ &= \frac{\mu u\cdots(\mu(u-\ell)+1)}{(\mu u+\kappa)\cdots(\mu(u-\ell)+\kappa+1)} \\ &\le \left(\frac{\mu u}{\mu u+\kappa}\right)^{\mu\cdot\ell} . \end{aligned} \tag{4.1}$$

In particular, if $\ell \ge cu$ for some $c \in [0,1]$, then, assuming $\kappa \le \frac{\mu u}{2}$,

$$P_{\text{c\&c}}(\mu,u,\kappa,\ell) \le \left(\frac{\mu u}{\mu u+\kappa}\right)^{\mu uc} = \left(\frac{1}{1+\frac{\kappa}{\mu u}}\right)^{\mu uc} = \frac{1}{\left(1+\frac{\kappa}{\mu u}\right)^{\frac{\mu u}{\kappa}\cdot c\kappa}} \le \frac{1}{2^{c\kappa}} . \tag{4.2}$$

Probability of passing pairwise checking. For the pairwise checking, we partition the μu tuples into u groups of size μ, so that only one tuple of each group is left after checking. We have $\binom{\mu u}{\mu}$ ways to select the first group, $\binom{\mu u-\mu}{\mu}$ ways to select the second group, $\binom{\mu u-2\mu}{\mu}$ ways to select the third group, etc. If we multiply all these values, we get the number of all possible groupings, where the order of the groups matters. Since the order of the groups is not important, we have to divide the final number by $u!$. The number of groupings of μu tuples into u groups is

$$\begin{aligned} \mathcal{G}(\mu,u) &= \frac{1}{u!}\prod_{i=0}^{u-1}\binom{\mu(u-i)}{\mu} \\ &= \frac{1}{u!}\left(\frac{1}{\mu!}\right)\prod_{i=0}^{u-1}\frac{(\mu(u-i))!}{(\mu(u-i-1))!} \\ &= \frac{1}{u!}\left(\frac{1}{\mu!}\right)^{u}\frac{(\mu(u-0))!}{(\mu(u-(u-1)-1))!} \\ &= \frac{(\mu u)!}{u!(\mu!)^u} . \end{aligned} \tag{4.3}$$

In order to pass the pairwise checking, all the $\mu\ell$ bad tuples should form exactly ℓ groups of size μ, such that no "good" tuple is present in any of these groups. The number of such groupings is $\mathcal{G}(\mu,\ell)\cdot\mathcal{G}(\mu,u-\ell)$, and thus the probability of

passing the pairwise check is

$$\begin{aligned} P_{\mathrm{pwc}}(\mu,u,\ell) &= \frac{\mathcal{G}(\mu,\ell)\cdot\mathcal{G}(\mu,u-\ell)}{\mathcal{G}(\mu,u)} \\ &= \frac{(\mu\ell)!}{\ell!(\mu!)^{\ell}}\cdot\frac{(\mu(u-\ell))!}{(u-\ell)!(\mu!)^{(u-\ell)}}\cdot\frac{u!(\mu!)^{u}}{(\mu u)!} \\ &= \frac{u!}{\ell!(u-\ell)!}\cdot\frac{(\mu\ell)!(\mu u-\mu\ell)!}{(\mu u)!} \\ &= \binom{u}{\ell}/\binom{\mu u}{\mu\ell} \ . \end{aligned} \tag{4.4}$$

Probability of passing both checks. This is the product of (4.1) and (4.4):

$$\begin{aligned} P_{\mathrm{pp}}(\mu,u,\kappa,\ell) &= \frac{\binom{\mu(u-\ell)+\kappa}{\kappa}\binom{u}{\ell}}{\binom{\mu u+\kappa}{\kappa}\binom{\mu u}{\mu\ell}} \\ &= \binom{u}{\ell}\cdot\frac{(\mu u-\mu\ell+\kappa)!}{(\mu u-\mu\ell)!\kappa!}\cdot\frac{(\mu u)!\kappa!}{(\mu u+\kappa)!}\cdot\frac{(\mu\ell)!(\mu u-\mu\ell)!}{(\mu u)!} \\ &= \binom{u}{\ell}/\binom{\mu u+\kappa}{\mu\ell} \ . \end{aligned} \tag{4.5}$$

Catching a single bad tuple. Suppose that the malicious prover aims to have a single bad tuple among the final u ones, i.e. $\ell = 1$. In this case

$$P_{\mathrm{pp}}(\mu,u,\kappa,1) = \binom{u}{1}/\binom{\mu u+k}{\mu\cdot 1} \le u^{-1}\left(u+\frac{k}{\mu}\right)^{\mu} \le u^{1-\mu} \ .$$

In order to have $P_{\mathrm{pp}}(\mu,u,\kappa,1) \le 2^{-\eta}$, it is sufficient to have $u^{1-\mu} \le 2^{-\eta}$ or $\mu \ge 1+\eta/\log u$.

Making a single bad tuple the worst case. We aim to select the parameters μ and κ in such a way, that aiming for a single bad tuple is the best strategy for the malicious prover.

First, let $\ell < cu$ for some $c \in [0,1]$ (fixed below). We consider the ratio $P_{\mathrm{pp}}(\mu,u,\kappa,\ell)/P_{\mathrm{pp}}(\mu,u,\kappa,\ell+1)$ and search for sufficient conditions for it to be larger than 1. Let $a^{\underline{n}} := a(a-1)\cdots(a-n+1)$. An upper bound for the ratio is

$$\begin{aligned} \frac{P_{\mathrm{pp}}(\mu,u,\kappa,\ell)}{P_{\mathrm{pp}}(\mu,u,\kappa,\ell+1)} &= \frac{\binom{u}{\ell}\binom{\mu u+\kappa}{\mu\ell+\mu}}{\binom{u}{\ell+1}\binom{\mu u+\kappa}{\mu\ell}} \\ &= \frac{(\ell+1)!(u-\ell-1)!}{(u-\ell)!\ell!}\cdot\frac{(\mu u-\mu\ell+\kappa)!(\mu\ell)!}{(\mu u+k-\mu\ell-\mu)!(\mu\ell+\mu)!} \\ &= \frac{(\ell+1)}{(u-\ell)}\cdot\frac{(\mu(u-\ell)+\kappa)^{\underline{\mu}}}{(\mu(\ell+1))^{\underline{\mu}}} \\ &\ge \frac{1}{u}\cdot\left(\frac{\mu(u-\ell-1)+\kappa+1}{\mu\ell+1}\right)^{\mu} \ . \end{aligned} \tag{4.6}$$

For (4.6) to be at least 1, it is sufficient to take

$$\mu(u-\ell-1)+\kappa+1 \geq u^{1/\mu}(\mu\ell+1),$$

meaning that it suffices for κ to be at least

$$\begin{aligned} u^{1/\mu}(\mu\ell+1)-1-\mu(u-\ell-1) &= \mu(u^{1/\mu}\ell-u+\ell)+u^{1/\mu}+\mu-1 \\ &\leq \mu(u^{1/\mu}cu-u+cn)+u^{1/\mu}+\mu-1 \\ &= \mu u(c(u^{1/\mu}+1)-1)+u^{1/\mu}+\mu-1 \ . \end{aligned}$$

If we take $c=1/(u^{1/\mu}+1)$, then this quantity is upper bounded by $u^{1/\mu}+\mu-1$, which is a sufficient choice for κ.

Now let $\ell \geq cu$. In this case, by (4.2), already the probability of passing cut-and-choose is less than 2^{-ck}, on the condition $k \leq \frac{\mu u}{2}$. It is sufficient to take $k \geq \eta/c = \eta(u^{1/\mu}+1)$ for this probability to be smaller than $2^{-\eta}$.

Due to the condition $k \leq \frac{\mu u}{2}$, we need to show that $\eta(u^{1/\mu}+1) \leq \frac{\mu u}{2}$, and $u^{1/\mu}+\mu-1 \leq \frac{\mu u}{2}$. We have shown that, for catching a single tuple, we should anyway choose $\mu \geq 1+\eta/\log u$. We get

$$\eta(u^{1/\mu}+\mu-1) \leq u^{1/(1+\eta/\log n)}+\mu-1 \leq u^{\log n/\eta}+\mu-1 = 2^{-\eta}+\mu-1 \leq \mu \leq \frac{\mu u}{2}$$

for $u \geq 2$, and

$$\eta(u^{1/\mu}+1) \leq \eta(u^{1/(1+\eta/\log n)}+1) \leq \eta(u^{\log n/\eta}+1) = \eta(2^{-\eta}+1) \leq \frac{3}{2}\eta \ .$$

In order to get $\frac{\mu u}{2} \geq \frac{3\eta}{2}$, we need $\mu \geq \frac{3\eta}{u}$. Since $\mu \geq 1+\eta/\log u > \eta/\log u$, it suffices to have $\log u \leq \frac{u}{3}$, which is true for $u \geq 12$. Such a choice for u is reasonable, since we may always generate more tuples than we actually need, and the preprocessing phase is in general run in advance for several protocol executions.

Summary. In order to allow a bad tuple pass with the probability of at most $2^{-\eta}$, while ending up with u tuples, it is sufficient to choose the parameters μ and κ as follows:

$$\begin{aligned} &\mu \geq 1+\eta/\log u \ , \\ &\kappa \geq \max\{(u^{1/\mu}+1)\eta, u^{1/\mu}+\mu-1\} \ . \end{aligned}$$

This choice of μ and κ is given for each type of tuples separately. If the total number of generated tuples is N, then it suffices in any case to take $\mu \geq 1+\eta/\log N$ and $\kappa \geq \max\{(N^{1/\mu}+1)\eta, N^{1/\mu}+\mu-1\}$. □

4.5.5 Verification of Circuit Computation

The ideal functionality $\mathcal{F}_{verify}$ for verification of circuit computation has been given in Figure 4.2. It allows to verify computations w.r.t. committed inputs, outputs, randomness, and communicated messages. Figure 4.30 depicts essentially the same functionality, but it uses a different notation. In particular, instead of taking circuits directly as inputs, it takes a set functions that are going to be verified. Functional representation makes handling of identifiers easier in the proofs.

The protocol Π_{verify} implementing it is given in Figure 4.31-4.33. It works on top of the commitment functionality $\mathcal{F}_{commit}$ defined in Section 4.5.2, the precomputed tuple generation functionality $\mathcal{F}_{pre}$ defined in Section 4.5.4, and the randomness generation functionality $\mathcal{F}_{rnd}$ defined in Section 4.5.3. Here $\mathcal{F}_{commit}$ is an inner subroutine of Π_{verify} that is global w.r.t. $\mathcal{F}_{pre}$ and $\mathcal{F}_{rnd}$. In this way, we may prove security of Π_{verify} in the ordinary UC model. When using $\mathcal{F}_{rnd}$ and $\mathcal{F}_{pre}$ as subroutines, we must remember that the protocols Π_{rnd} and Π_{pre} implementing them work under certain assumptions about the interaction of $\mathcal{Z}$ and $\underline{\mathcal{F}_{commit}}$. This interaction is now fully covered by Π_{verify}, and we may now leave $\mathcal{Z}$ unconstrained. The interaction of Π_{verify} with $\mathcal{F}_{commit}$ does satisfy the condition that no inner identifiers of Π_{pubrnd} and Π_{pre} should be accessed. Namely, since any invocation of $\mathcal{F}_{commit}$ is possible only if all honest parties agree on it, and Π_{verify} even does not see the inner identifiers of Π_{rnd} and Π_{pre} implementing $\mathcal{F}_{rnd}$ and $\mathcal{F}_{pre}$, the honest parties clearly do not use these identifiers. Without approval of the honest parties, there is no way for corrupted parties to open any inner commitments of Π_{pre} and Π_{rnd}.

Both $\mathcal{F}_{verify}$ and Π_{verify} use unique identifiers id. Each such identifier corresponds to some wire of the circuit that is being verified. It encodes the two parties $p(id)$ and $p'(id)$ (possibly $p(id) = p'(id)$) committed to a particular valuation $comm[id]$ of the wire, and a function $f(id) =: f'$ (a composition of basic circuit operations given in Section 4.2) with its input identifiers $\vec{xid}(id)$, so that the parties may verify whether $comm[id] = f'((comm[i])_{i \in \vec{xid}(id)})$. If the computation of $comm[id]$ is not needed to be verified (i.e. it is some input commitment), then formally. $f(id) = \mathsf{id}_R$ (identity over some ring R), and $\vec{xid}(id) = []$.

Observation 4.5. From the definition of Π_{verify}, we extract the number of different tuples required for each operation type directly from the description of the initialization phase. They are given in Table 4.5.

Lemma 4.7 (cost of initializing Π_{verify})**.** *Let Π_{verify} use the implementation Π_{pre} of $\mathcal{F}_{pre}$ with λ-bit randomness seed, and the parameters μ, κ. Let all the functions f to be verified consist of basic operations f_i, such that there are N_b operations requiring bit decompositions (bit decomposition, ring extension), and N_m multiplications. Let 2^m be the cardinality of the largest ring involved in the computation.*

$\mathcal{F}_{verify}$ works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ committed to $comm[id]$, the function $f(id)$ to verify, and the input identifiers $\vec{xid}(id)$ on which $f(id)$ should be verified w.r.t. the output identified by id. It also encodes the randomness $r(id)$ that is generated during the initialization, and will be used later for the randomness commitment. The messages are first stored in an array $sent$ before the sender and the receiver get finally committed to them.

• **Initialization:** On input $(\mathsf{init}, \hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'})$ from all (honest) parties, where $\hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'}$ are defined over the same domain, assign $f \leftarrow \hat{f}$, $\vec{xid} \leftarrow \hat{\vec{xid}}$, $p \leftarrow \hat{p}$, $p' \leftarrow \hat{p'}$. For all $id \in \mathsf{Dom}(f)$, generate a fresh randomness $r(id)$ in $\mathbb{Z}_{2^m}$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. For $p(id) \in \mathcal{C}$, deliver $r(id)$ to $\mathcal{A}_S$. Deliver $(\mathsf{init}, f, \vec{xid}, p, p')$ to $\mathcal{A}_S$. If $\mathcal{A}_S$ responds with (stop), output $\perp$ to all parties.

• **Input Commitment:** On input $(\mathsf{commit\_input}, id, x)$ from $P_{p(id)}$, and $(\mathsf{commit\_input}, id)$ from all (honest) parties, if $comm[id]$ it not defined yet, assign $comm[id] \leftarrow x$. If $p(id) \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$.

• **Message Commitment:** On input $(\mathsf{send\_msg}, id, x)$ from $P_{p(id)}$ and $(\mathsf{send\_msg}, id)$ from all (honest) parties, output x to $P_{p'(id)}$. If $p(id) \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S$. If $p'(id) \in \mathcal{C}$, output x to $\mathcal{A}_S$. Assign $sent[id] \leftarrow x$.

On input $(\mathsf{commit\_msg}, id)$ from all (honest) parties, check if $sent[id]$ and $comm[id]$ are defined. If $sent[id]$ is defined, and $comm[id]$ is not defined, assign $comm[id] \leftarrow sent[id]$. If both $p(id), p'(id) \in \mathcal{C}$, assign $comm[id] \leftarrow x^*$, where x^* is chosen by $\mathcal{A}_S$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$ from $P_{p(id)}$, and $(\mathsf{commit\_rnd}, id)$ from all (honest) parties, if $comm[id]$ is not defined yet, assign $comm[id] \leftarrow r(id)$.

• **Verification:** On input (verify, id) from all (honest) parties, if $comm[id]$ and $comm[i]$ have been defined for all $i \in \vec{xid}(id)$, take $\vec{x} \leftarrow (comm[i])_{i \in \vec{xid}(id)}$ and $y \leftarrow comm[id]$. For $f \leftarrow f(id)$, compute $y' \leftarrow f(\vec{x})$. If $y' - y = 0$, output $(id, 1)$ to each party. Otherwise, output $(id, 0)$ to each party. Output the difference $y' - y$, to $\mathcal{A}_S$.

• **Cheater detection:** On all inputs involving id, if $p(id) \in \mathcal{C}$, $\mathcal{A}_S$ may input $(\mathsf{cheater}, p(id))$. In this case, $comm[id]$ and $sent[id]$ are not assigned, $(\mathsf{cheater}, p(id))$ is output to each party, and $\mathcal{F}_{verify}$ assigns $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$. If $\mathcal{A}_S$ does not input $(\mathsf{cheater}, p(id))$, then each commitment ends up outputting $(\mathsf{confirmed}, id)$ to each party.

If $(\mathsf{cheater}, p(id))$ comes from $\mathcal{A}_S$ during the execution of (verify, id), then $\mathcal{F}_{verify}$ outputs $(id, 0)$ to all parties instead of $(\mathsf{cheater}, p(id))$.

Figure 4.30: Ideal functionality $\mathcal{F}_{verify}$

Table 4.5: Number of precomputed tuples needed for basic operations

operation	tuple type	# tuples	bits per tuple
Linear combination	–	–	–
Conversion to a smaller ring	–	–	–
Bit decomposition in $\mathbb{Z}_{2^m}$	bit	m	m
Multiplication in $\mathbb{Z}_{2^m}$	triple	1	m
Extending $\mathbb{Z}_{2^{m_x}}$ to $\mathbb{Z}_{2^{m_y}}$	bit	m_x	m_y

In Π_{verify}, each party works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ committed to $comm[id]$, the function $f(id)$ to verify, and the identifiers $\vec{xid}(id)$ of the inputs on which $f(id)$ should be verified w.r.t. the output identified by id. It also encodes the randomness $r(id)$ that is be precomputed by the parties during the initialization. The prover stores the committed values in a local array $comm$. The verifiers store the helpful values published by the verifier in an array $pubv$. The messages are stored by the sender and the receiver in a local array $sent$ before they finally get committed to these messages. Π_{verify} uses $\mathcal{F}_{transmit}$, $\mathcal{F}_{commit}$, $\mathcal{F}_{rnd}$, and $\mathcal{F}_{pre}$ as subroutines.

• **Initialization:** On input $(\mathsf{init}, \hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'})$, where domains of the mappings $\hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p}'$ are the same, initialize $comm$ and $sent$ to empty arrays. Assign $f \leftarrow \hat{f}, \vec{xid} \leftarrow \hat{\vec{xid}}, p \leftarrow \hat{p}, p' \leftarrow \hat{p'}$.

Initialize subroutine protocols:

- *Initialize* $\mathcal{F}_{transmit}$: For all $id \in \mathsf{Dom}(f)$ s.t $p(id) \neq p'(id)$, define the mappings s, r, f' such that $s(id) \leftarrow p(id)$, $r(id) = f'(id) \leftarrow p'(id)$. For all $i \in [n]$, define an identifier $id' \leftarrow (\mathsf{bc}, i)$ that will be used for broadcast, and $s(id) \leftarrow i$, $r(id) \leftarrow \perp$, $f'(id) \leftarrow \perp$. Send $(\mathsf{init}, s, r, f')$ to $\mathcal{F}_{transmit}$.
- *Initialize* $\mathcal{F}_{pre}$: A message $(\mathsf{init}, \bar{m}, \bar{u}, \bar{p})$ is sent to $\mathcal{F}_{pre}$, where $\bar{m}$,$\bar{u}$,$\bar{p}$ depend on the functions f to be verified. Let $f(id)$ be a composition of basic operations $f_1, \ldots, f_{N_{id}}$. Each such f_i, introduces to $\mathcal{F}_{pre}$ identifiers of the form $id' \leftarrow (id_i, type, m, u)$ such that $type$ is the type of the tuple, $\bar{m}(id') = m$, $\bar{u}(id') = u$. For all id', take $\bar{p}(id') \leftarrow p(id)$.
  1. *Linear combination, conversion to a smaller ring:* no tuples needed;
  2. *Bit decomposition in* $\mathbb{Z}_{2^m}$: $(id_i, \mathsf{bit}, m, m)$;
  3. *Multiplication in* $\mathbb{Z}_{2^m}$*:* $(id_i, \mathsf{triple}, m, 1)$;
  4. *Extending* $\mathbb{Z}_{2^{m_x}}$ *to a larger ring* $\mathbb{Z}_{2^{m_y}}$*:* $(id_i, \mathsf{bit}, m_y, m_x)$.

  Let pre be the array containing all such identifiers introduced by all basic operations of $f(id)$. Since $\mathcal{F}_{pre}$ generates all the tuples of the same type simultaneously, the tuple generation is optimized by substituting all the identifiers $(id_i, type, m, u_{id_i})$ for the same $type$ and bit size m with a single identifier $id' = (type, m, u)$ for $u = \sum u_{id_i}$. After inducing $\bar{m}, \bar{u}, \bar{p}$ from these new identifiers, each party sends $(\mathsf{init}, \bar{m}, \bar{u}, \bar{p})$ to $\mathcal{F}_{pre}$.
- *Initialize* $\mathcal{F}_{commit}$: For commitments of non-random wires, take $\tilde{p}(id) \leftarrow p(id)$, and $\tilde{m}(id) \leftarrow m$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. If $p(id) \neq p'(id)$, generate a new identifier id' and define additionally $\tilde{m}(id') \leftarrow m(id)$, $\tilde{p}(id') \leftarrow p'(id)$. After doing it for all id, send $(\mathsf{init}, \tilde{m}, \tilde{p})$ to $\mathcal{F}_{commit}$.
- *Initialize* $\mathcal{F}_{rnd}$: For commitments of random wires, take $\tilde{p}(id) \leftarrow p(id)$, and $\tilde{m}(id) \leftarrow m$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. After doing it for all id, send $(\mathsf{init}, \tilde{m}, \tilde{p})$ to $\mathcal{F}_{rnd}$.

Generating precomputed tuples and randomness: A message $(type, id')$ is sent to $\mathcal{F}_{pre}$ by each party for each identifier $id' = (type, m, u)$ on which $\mathcal{F}_{pre}$ was initialized. $\mathcal{F}_{pre}$ generates the required tuples and automatically copies them to $\mathcal{F}_{commit}$.
Similarly, (rnd, id') is sent to $\mathcal{F}_{rnd}$ by each party P_j for each id' on which $\mathcal{F}_{rnd}$ was initialized. $\mathcal{F}_{rnd}$ generates the randomness and automatically copies it to $\mathcal{F}_{commit}$.

Initialization failure: If $(id, \perp)$ comes from $\mathcal{F}_{pre}$ of $\mathcal{F}_{rnd}$ for some id, then output $\perp$ to $\mathcal{Z}$.

• **Cheater detection:** At any time, when $\mathcal{F}_{transmit}$ or $\mathcal{F}_{commit}$ outputs a message $(\mathsf{cheater}, k)$, output $(\mathsf{cheater}, k)$ to $\mathcal{Z}$. Treat P_k as if it has left the protocol, i.e. assign $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$. If $(\mathsf{cheater}, p(id))$ comes from $\mathcal{F}_{commit}$ during executing (verify, id), then all parties output $(id, 0)$ instead of $(\mathsf{cheater}, p(id))$, denoting that the proof of $p(id)$ has failed.

Figure 4.31: Real protocol Π_{verify} (initialization, cheater detection)

• **Input Commitment:** On input $(\mathsf{commit\_input}, id, x)$, $P_{p(id)}$ sends (commit, id, x) to $\mathcal{F}_{commit}$. On input $(\mathsf{commit\_input}, id)$, each party sends (commit, id) to $\mathcal{F}_{commit}$. After getting $(\mathsf{confirmed}, id)$ from $\mathcal{F}_{commit}$, $P_{p(id)}$ assigns $comm[id] \leftarrow x$, and each other party outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Message Commitment:**

1. On input $(\mathsf{send\_msg}, id, x)$, $P_{p(id)}$ sends $(\mathsf{transmit}, id, x)$ to $\mathcal{F}_{transmit}$. On input $(\mathsf{send\_msg}, id)$, $P_{p'(id)}$ waits for (id, x) from $\mathcal{F}_{transmit}$. If the transmission succeeds, both parties assign $sent[id] \leftarrow x$.
2. On input $(\mathsf{commit\_msg}, id)$, $P_{p(id)}$ and $P_{p'(id)}$ send to $\mathcal{F}_{commit}$ the message $(\mathsf{mcommit}, id, id', sent[id])$. Each other party sends $(\mathsf{mcommit}, id, id')$ to $\mathcal{F}_{commit}$. The identifier id' has been defined for this particular id in the initialization phase in such a way that the party committed to $comm[id']$ of $\mathcal{F}_{commit}$ is $P_{p'(id)}$.

If $(id, id', \perp)$ is output by $\mathcal{F}_{commit}$, then each party sends (reveal, id) to $\mathcal{F}_{transmit}$, and after getting back (id, x), it sends $(\mathsf{pcommit}, id, x)$ and $(\mathsf{pcommit}, id', x)$ to $\mathcal{F}_{commit}$. After getting $(\mathsf{confirmed}, id)$ from $\mathcal{F}_{commit}$, $P_{p(id)}$ and $P_{p(id')}$ assign $comm[id] \leftarrow sent[id]$, and each other party outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$, $P_{p(id)}$ takes $comm[id] \leftarrow r(id)$ for the previously generated randomness $r(id)$. Each party outputs $(\mathsf{confirmed}, id)$ to $\mathcal{Z}$.

• **Verification:** On input (verify, id), each party P_k checks whether $comm[i]$ has been defined for all $i \in \vec{xid}(id)$. It decomposes $f(id)$ to basic operations $f_1, \ldots, f_N$. For each f_i, some additional identifiers are defined: $id_i^{\mathsf{x}k}$ for the k-th input, $id_i^{\mathsf{y}k}$ for the k-th output, and $id_i^{\mathsf{z}k}$ for the k-th alleged zero of f_i (some of these will actually overlap). The index k is omitted if there is only one such identifier. The symbols other than x, y, z are used for intermediate values. Let $id_{i,k}^{type}$ be the index of $\mathcal{F}_{commit}$ storing the k-th component of the i-th tuple of type $type$ generated by $\mathcal{F}_{pre}$.

First, $P_{p(id)}$ computes all the intermediate values $comm[id_i^{\mathsf{x}k}]$ using the function descriptions f_i and the commitments $comm[i]$ for $i \in \vec{xid}(id)$. Let $\hat{\vec{x}} = [\hat{\vec{x}}_1 \| \cdots \| \hat{\vec{x}}_N]$ denote all values that should be broadcast by $P_{p(id)}$, where $\hat{\vec{x}}_i$ is determined by the operation f_i, its inputs $comm[id_i^{\mathsf{x}k}]$, and the precomputed tuples $comm[id_{i,k}^{type}]$:

1. *Linear combination*: no broadcasts.
2. *Multiplication in* $\mathbb{Z}_{2^m}$: $\hat{\vec{x}}_i \leftarrow [(x_1 - a) \bmod 2^m, (x_2 - b) \bmod 2^m]$, where $a = comm[id_{i,1}^{\mathsf{triple}}]$, $b = comm[id_{i,2}^{\mathsf{triple}}]$, $x_1 = comm[id_i^{\mathsf{x}1}]$, $x_2 = comm[id_i^{\mathsf{x}2}]$.
3. *Bit decomposition in* $\mathbb{Z}_{2^m}$: $\hat{\vec{x}}_i \leftarrow [c_1, \ldots, c_m]$, where $c_k \in \{0, 1\}$ denotes whether the trusted bit $comm[id_{i,k}^{\mathsf{bit}}]$ is different from the k-th bit of $comm[id_i^{\mathsf{x}}]$.
4. *Conversion to a smaller ring*: no broadcasts.
5. *Conversion from* $\mathbb{Z}_{2^{m_x}}$ *to a larger ring* $\mathbb{Z}_{2^{m_y}}$: Perform bit decomposition of $comm[id_i^{\mathsf{x}}]$ over $\mathbb{Z}_{2^{n_y}}$, getting n_y bits b_k. Take the first n_x of these bits.
   $\hat{\vec{x}}_i \leftarrow [c_1, \ldots, c_{n_x}]$, where $c_k \in \{0, 1\}$ denotes whether the trusted bit $comm[id_{i,k}^{\mathsf{bit}}]$ is different from b_k.

$P_{p(id)}$ sends $(\mathsf{broadcast}, (\mathsf{bc}, p(id)), (id_i^{type}, \hat{\vec{x}}_i)_{i \in [N]})$ to $\mathcal{F}_{transmit}$. Upon receiving $(\mathsf{broadcast}, (\mathsf{bc}, p(id)), (id_i^{type}, \hat{\vec{x}}_i)_{i \in [N]})$, each party writes $pubv[id_i^{type}] \leftarrow \hat{\vec{x}}_i$. For simplicity of further verifications, we assume that $(id_{i,k}^{\mathsf{bit}} = 1 - id_{i,k}^{\mathsf{bit}})$ is immediately sent to $\mathcal{F}_{commit}$ for all k such that $c_k = 1$ was broadcast, so that we do not need to compute it for each operation separately.

Figure 4.32: Real protocol Π_{verify} (commitments, 1^{st} step of verification)

After the verifiers have assigned $pubv[id_i^{type}]$, they start computing f_i, for all $i \in [N]$. The basic operations f_i are computed by sending lc and trunc to $\mathcal{F}_{commit}$. The outputs of f_i are stored in $\mathcal{F}_{commit}$ under identifiers id_i^{y} (used also as $id_{i'}^{\mathsf{x}}$ for computing the next basic operations $f_{i'}$ for $i' > i$), and the alleged zeroes are stored in $\mathcal{F}_{commit}$ under identifiers id_i^{z}.

1. *Linear combination* $[c_0, \ldots, c_l]$: Send $(id_i^{y} = c_0 + \sum_{k\in[l]} c_k \cdot id_i^{x_k})$ to $\mathcal{F}_{commit}$.
2. *Multiplication in* $\mathbb{Z}_{2^m}$: Let $(id^a, id^b, id^c) = (id_{i,k}^{\mathsf{triple}})_{k\in[3]}$, and $[\hat{x}_1, \hat{x}_2] = pubv[id_i^{\mathsf{triple}}]$. Send to $\mathcal{F}_{commit}$:
   - $id_i^{\mathsf{y}} = \hat{x}_1 \cdot \hat{x}_2 + \hat{x}_1 \cdot id^b + \hat{x}_2 \cdot id^a + id^c$;
   - $id_i^{\mathsf{z}_1} = \hat{x}_1 + id^a - id_i^{\mathsf{x}_1}$;
   - $id_i^{\mathsf{z}_2} = \hat{x}_2 + id^b - id_i^{\mathsf{x}_2}$.
3. *Bit decomposition in* $\mathbb{Z}_{2^m}$: Let $[id^{b_1}, \ldots, id^{b_m}] = (id_{i,k}^{\mathsf{bit}})_{k\in[m]}$.
   Send $(id_i^{\mathsf{z}} = id_i^{\mathsf{x}} - \sum_{k=1}^{m} 2^{k-1} \cdot id^{b_k})$ to $\mathcal{F}_{commit}$.
4. *Conversion to a smaller ring* $\mathbb{Z}_{2^m}$: Send $(id_i^{\mathsf{y}} = id_i^{\mathsf{x}} \bmod 2^m)$ to $\mathcal{F}_{commit}$.
5. *Conversion from* $\mathbb{Z}_{2^{m_x}}$ *to a larger ring* $\mathbb{Z}_{2^{m_y}}$: Let $[id^{b_1}, \ldots, id^{b_{m_x}}] = (id_{i,k}^{\mathsf{bit}})_{k\in[m_x]}$. Send to $\mathcal{F}_{commit}$
   - $id_i^{\mathsf{y}} = \sum_{k=1}^{m_x} 2^{k-1} \cdot id^{b_k}$;
   - $id_i^{\mathsf{w}} = id_i^{\mathsf{y}} \bmod m_x$;
   - $id_i^{\mathsf{z}} = id_i^{\mathsf{x}} - id_i^{\mathsf{w}}$.

Send also $(id_{N+1}^{\mathsf{z}_k} = id_N^{\mathsf{y}_k} - id)$ to $\mathcal{F}_{commit}$, to verify the final output against $comm[id]$.
After all f_i have been processed, for each alleged zero $id_i^{\mathsf{z}_k}$, each party first sends $(\mathsf{weak\_open}, id_i^{\mathsf{z}_k})$ to $\mathcal{F}_{commit}$. If $\mathcal{F}_{commit}$ outputs $(id, \perp)$, then each party sends $(\mathsf{open}, id_i^{\mathsf{z}_k})$ to $\mathcal{F}_{commit}$. Upon receiving all $(id_i^{\mathsf{z}_k}, z_{ik})$ from $\mathcal{F}_{commit}$, if $z_{ik} = 0$ for all i, k, then an honest party outputs 1. Otherwise, it outputs 0.

Figure 4.33: Real protocol Π_{verify} (2^{nd} step of verification)

The cost of initializing Π_{verify} is upper bounded by

$$\begin{aligned}\mathsf{vcost}_{pre}^{N_b,N_m,m} &= \mathsf{tr}_{\mathsf{sh}_n\cdot m\cdot((\mu(N_b\cdot m+3N_m)+\kappa(m+3))}^{\otimes n} \otimes (\mathsf{tr}_{\mathsf{sh}_n\cdot\lambda}^{\otimes nt} \oplus \mathsf{bc}_{\mathsf{sh}_n\cdot\lambda}^{\otimes n}) \\ &\quad \oplus (\mathsf{bc}_{\mathsf{sh}_n\cdot m\cdot\kappa(m+3)}^{\otimes n} \otimes \mathsf{bc}_{(\mu-1)m(N_b+\mathsf{sh}_n\cdot 2N_m)}^{\otimes n} \\ &\quad \otimes \mathsf{bc}_{\mathsf{sh}_n\cdot(\mu-1)m(N_b m+N_m)}^{\otimes n}) \ .\end{aligned}$$

Proof. The cost comes from the generation of precomputed tuples. The number of different tuples used by each operation is given in Table 4.5. By Lemma 4.5, the cost of generating N tuples of type x over a ring of size 2^m is

$$\begin{aligned}\mathsf{prc}_T^N &= \mathsf{tr}_{(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}}T)}^{\otimes n} \otimes (\mathsf{tr}_{\mathsf{sh}_n\cdot\lambda}^{\otimes nt} \oplus \mathsf{bc}_{\mathsf{sh}_n\cdot\lambda}^{\otimes n}) \\ &\quad \oplus (\mathsf{bc}_{\kappa\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}}T)}^{\otimes n} \otimes \mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_1}T)}^{\otimes n} \otimes \mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_2}T)}^{\otimes n}) \ .\end{aligned}$$

where $T = (x, m)$, and the definitions of subterms can be found in Table 4.4.

In this way, the total number of the transmitted and broadcast bits is linear in the terms $N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{tpl}}(x, m)$, $N \cdot \mathsf{nb}_{\mathsf{op}_1}(x, m)$, and $N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(x, m)$. Hence it suffices to find the upper bounds for these three quantities. We use the fact that the share overhead is linear w.r.t. the number of shared bits, i.e. $\mathsf{sh}_n \cdot (M_1 + M_2) = \mathsf{sh}_n \cdot M_1 + \mathsf{sh}_n \cdot M_2$ (see Observation 4.2).

- The bit decomposition and the conversion to a larger ring both require m trusted bits of m bits each. As shown in the proof of Lemma 4.5, the multiplier sh_n can be removed from $\mathsf{nb}_{\mathsf{op}_1}(\mathsf{bit}, m)$ since the prover broadcasts the difference bit in plain. Hence for the bit-related gates we have

$$\begin{aligned}N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{tpl}}(x, m) &\leq (N_b \cdot m) \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{tpl}}(\mathsf{bit}, m) \\ &= N_b \cdot \mathsf{sh}_n \cdot m^2 \ , \\ N \cdot \mathsf{nb}_{\mathsf{op}_1}(x, m) &\leq (N_b \cdot 1) \cdot \mathsf{nb}_{\mathsf{op}_1}(\mathsf{bit}, m) \\ &= N_b \cdot m \ , \\ N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(x, m) &\leq (N_b \cdot m) \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(\mathsf{bit}, m) \\ &= N_b \cdot \mathsf{sh}_n \cdot m^2 \ .\end{aligned}$$

- Each multiplication gate requires one multiplication triple. Hence for the multiplication gates we have

$$\begin{aligned}N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{tpl}}(x, m) &\leq N_m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{tpl}}(\mathsf{triple}, m) \\ &= N_m \cdot \mathsf{sh}_n \cdot 3m \ , \\ N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_1}(x, m) &\leq N_m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_1}(\mathsf{triple}, m) \\ &= N_m \cdot \mathsf{sh}_n \cdot 2m \ , \\ N \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(x, m) &\leq N_m \cdot \mathsf{sh}_n \cdot \mathsf{nb}_{\mathsf{op}_2}(\mathsf{triple}, m) \\ &= N_m \cdot \mathsf{sh}_n \cdot m \ .\end{aligned}$$

Table 4.6: Number of $\mathcal{F}_{commit}$ and $\mathcal{F}_{transmit}$ operations needed for committing M values of N bits each in $\mathcal{F}_{verify}$

commitment	call	# calls	total $\mathcal{F}_{transmit}$ cost
commit_input	commit	1	$\mathsf{tr}^{\otimes nM}_{\mathsf{sh}_n \cdot N}$
send_msg	transmit	1	$\mathsf{tr}^{\otimes M}_{\mathsf{sh}_n \cdot N}$
commit_msg (cheap)	mcommit	1	$\mathsf{tr}^{\otimes nM}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{fwd}^{\otimes nM}_{\mathsf{sh}_n \cdot N}$
commit_msg (expensive)	reveal	1	$\mathsf{rev}^{\otimes M}_{\mathsf{sh}_n \cdot N}$
commit_rnd (during initialization)	commit	t	$\mathsf{tr}^{\otimes ntM}_{\mathsf{sh}_n \cdot N}$
	priv_open	t	$\mathsf{fwd}^{\otimes ntM}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{tr}^{\otimes ntM}_{\mathsf{sh}_n \cdot N}$

The randomness seed of λ bits may be generated once for all tuples. Different types of tuples can be generated in parallel. For simplicity, let μ and κ be the same for generating all types of tuples. Let x_1 be the total number of bits transmitted during the first round, when the initial unverified tuples are shared, and x_2, x_3, x_4 the total number of bits broadcast in the three parallel weak openings. The total cost is $(\mathsf{tr}^{\otimes n}_{x_1} \otimes f(\lambda)) \oplus (\mathsf{bc}^{\otimes n}_{x_2} \otimes \mathsf{bc}^{\otimes n}_{x_3} \otimes \mathsf{bc}^{\otimes n}_{x_4})$, where $f(\lambda) := \mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot \lambda} \oplus \mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot \lambda}$ does not depend on the tuple types. Upper bounds for x_i are

$$\begin{aligned}
x_1 &\leq (\mu N_b + \kappa) \cdot \mathsf{sh}_n \cdot m^2 + (\mu N_m + \kappa) \cdot \mathsf{sh}_n \cdot 3m \\
&= \mathsf{sh}_n \cdot m \cdot ((\mu N_b + \kappa) \cdot m + (\mu N_m + \kappa) \cdot 3) \\
&= \mathsf{sh}_n \cdot m \cdot ((\mu(N_b \cdot m + 3N_m) + \kappa(m+3)) \ , \\
x_2 &\leq \kappa \cdot \mathsf{sh}_n \cdot m^2 + \kappa \cdot \mathsf{sh}_n \cdot 3m \\
&= \mathsf{sh}_n \cdot m \cdot \kappa(m+3) \ , \\
x_3 &\leq (\mu - 1)(N_b \cdot m + N_m \cdot \mathsf{sh}_n \cdot 2m) \\
&= (\mu - 1)m(N_b + \mathsf{sh}_n \cdot 2N_m) \ , \\
x_4 &\leq (\mu - 1)(N_b \cdot \mathsf{sh}_n \cdot m^2 + N_m \cdot \mathsf{sh}_n \cdot m) \\
&= \mathsf{sh}_n \cdot (\mu - 1)m(N_b \cdot m + N_m) \ .
\end{aligned}$$

Substituting x_1, x_2, x_3, x_4 into $(\mathsf{tr}^{\otimes n}_{x_1} \otimes f(\lambda)) \oplus (\mathsf{bc}^{\otimes n}_{x_2} \otimes \mathsf{bc}^{\otimes n}_{x_3} \otimes \mathsf{bc}^{\otimes n}_{x_4})$, we get the value stated in this lemma. □

Observation 4.6. From the description of the commitment functions of Π_{verify}, we count the number of $\mathcal{F}_{transmit}$ and $\mathcal{F}_{commit}$ calls that it makes. They are given in Table 4.6. The cost of broadcasting the complaint (bad, k) is omitted.

Observation 4.7. By counting the number of the broadcast hint bits an the alleged zero bits for each basic operation, we get the numbers given in Table 4.7. Note that the hint bits c_i broadcast for each bit decomposition do not have to be committed in $\mathbb{Z}_{2^m}$, and each such bit is broadcast as a 1-bit value.

Table 4.7: Number of bits for verifying each operation in $\mathcal{F}_{verify}$

operation	# hint bits	# alleged zero bits
Linear comb., conversion to a smaller ring	0	0
Bit decomposition in $\mathbb{Z}_{2^m}$	m	m
Multiplication in $\mathbb{Z}_{2^m}$:	$2m$	$2m$
Extending $\mathbb{Z}_{2^{m_x}}$ to $\mathbb{Z}_{2^{m_y}}$	m_x	m_x

Lemma 4.8 (cost of the broadcasts of $\mathcal{F}_{verify}$)**.** *Let f be a function that is going to be verified. Let f consist of N basic operations $f_i \notin \{\mathsf{lc}, \mathsf{trunc}\}$. Let 2^m be the size of the largest used ring. The total cost of the broadcast phase of Π_{verify} is upper bounded by $\mathsf{bc}_{N \cdot 2m}$.*

Proof. All the bits are broadcast in parallel using $\mathcal{F}_{transmit}$. We use Table 4.7 to count the number of bits for each operation. We take the upper bound $2m$ on broadcast bits per operation, which comes from multiplication. Differently from the initialization phase of Π_{verify}, the costs are similar for distinct types of basic operations, as they are all $O(m)$. □

Lemma 4.9 (cost of the final verification of Π_{verify})**.** *Let all the functions f to be verified consist of N basic operations $f_i \notin \{\mathsf{lc}, \mathsf{trunc}\}$. Let M_y be the total number of bits output by f. Let M_x, M_r, M_c, M_{pre} be the total number of bits in the committed input, randomness, communicated elements, and precomputed tuples respectively. Let 2^m be the size of the largest used ring. The cost of the verification phase of Π_{verify} is upper bounded by:*

- $\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot (N \cdot 2m + M_y)}$, *if all* $(\mathsf{weak\_open}, id)$ *succeed,*
- $\mathsf{rev}^{\otimes (t-1)}_{\mathsf{sh}_n \cdot (M_x + M_r + M_c + M_{pre} + M_y)}$, *if some* $(\mathsf{weak\_open}, id)$ *outputs* $(id, \perp)$.

Proof. Taking into account the costs of different operations of Π_{commit} given in Observation 4.2, the functionalities lc and trunc do not take any communication. Hence the only cost for verifying different basic operations comes in the end, where the alleged zeroes are verified.

- Assume that $(\mathsf{weak\_open}, id)$ succeeds for all alleged zeroes. It has cost $\mathsf{bc}^{\otimes n}_M$ for an M-bit value. From Table 4.7, we see that the largest number of alleged zero checks per operation is $2m$ that comes from multiplication. In addition, there is an alleged zero bit for each of the M_y output bits of f. The broadcast is parallelizable, so all the bits are broadcast simultaneously.
- Assume that $(\mathsf{weak\_open}, id)$ returns $(id, \perp)$ for some alleged zero identifier id. In this case, open is used instead. For this, the parties should reveal

shares of all the initial inputs from which the alleged zeroes and the outputs were computed, and also the shares of the final outputs. If there is a conflict with each corrupted verifier, up to $t-1$ shares may have to be revealed, since there are at most $t-1$ corrupted parties. □

Lemma 4.10. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{verify} UC-realizes $\mathcal{F}_{verify}$ in $\mathcal{F}_{transmit}$-$\mathcal{F}_{commit}$-$\mathcal{F}_{pre}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{verify}$ described in Figure 4.34. It runs a local copy of Π_{verify}, together with local copies of $\mathcal{F}_{transmit}$, $\mathcal{F}_{commit}$, $\mathcal{F}_{pre}$, $\mathcal{F}_{rnd}$.

Simulatability. For the randomness commitments, $\mathcal{S}$ simulates $\mathcal{F}_{rnd}$. For this, it only needs to know $r(id)$ for $p(id) \in \mathcal{C}$, and in this case it gets $r(id)$ from $\mathcal{F}_{verify}$. For the commitment of precomputed tuples, $\mathcal{S}$ simulates $\mathcal{F}_{pre}$, and it does not need any additional information for this. If the initialization succeeds, $\mathcal{A}$ assumes that the randomness generated by $\mathcal{F}_{rnd}$ and the triples generated by $\mathcal{F}_{pre}$ are all copied to $\mathcal{F}_{commit}$, as defined in the descriptions of $\mathcal{F}_{pre}$ and $\mathcal{F}_{rnd}$ that are using shared $\underline{\mathcal{F}_{commit}}$.

For input and message commitments, $\mathcal{S}$ simulates $\mathcal{F}_{commit}$ and $\mathcal{F}_{transmit}$, where the inputs of dishonest parties are provided by $\mathcal{A}$, and the inputs of honest parties that should be delivered to a corrupted party are given to $\mathcal{S}$ by $\mathcal{F}_{verify}$.

For input commitments, if $p(id) \in \mathcal{C}$, the commitments may fail. In this case $\mathcal{S}$ delivers to $\mathcal{F}_{verify}$ the message $(\mathsf{cheater}, p(id))$.

For message commitments, $\mathcal{S}$ simulates sending $(\mathsf{mcommit}, id, id', x)$ and $(\mathsf{mcommit}, id, id', x')$ to $\mathcal{F}_{commit}$ by $P_{p(id)}$ and $P_{p'(id)}$ respectively. At this point, it either $p(id) \in \mathcal{C}$ or $p'(id) \in \mathcal{C}$, $\mathcal{S}$ knows both x and x' that these parties commit to. In this case, if $x \neq x'$, then $\mathcal{S}$ simulates $\mathcal{F}_{commit}$ outputting $(id, id', \perp)$. If either $(id, id', \perp)$, $(\mathsf{cheater}, p(id))$, or $(\mathsf{cheater}, p'(id))$ is output by $\mathcal{F}_{commit}$, then $\mathcal{S}$ simulates work of $\mathcal{F}_{transmit}$ on input (reveal, id). The latter results in opening (id, x) to $\mathcal{A}$, where x is the value that was actually transmitted, and since either $p(id) \in \mathcal{C}$ or $p'(id) \in \mathcal{C}$, this value is known to $\mathcal{S}$.

When the verification starts, $\mathcal{S}$ needs to simulate the broadcast, and it needs to generate the broadcast values of the honest provers itself. All of these values are some private values hidden by a random mask (each tuple is used only once), and hence are distributed uniformly. We discuss it in details for different kinds of tuples.

1. *Bit decomposition of x in $\mathbb{Z}_{2^m}$*: Since each b_k is distributed uniformly in $\mathbb{Z}_2$, the difference $b_k - x_k$ is also distributed uniformly in $\mathbb{Z}_2$.

2. *Multiplication of x_1 and x_2 in $\mathbb{Z}_{2^m}$*: Since the entries a and b of the triple (a, b, c) are distributed uniformly in $\mathbb{Z}_{2^m}$, so are the values $(x_1 - a) \bmod 2^m$ and $(x_2 - b) \bmod 2^m$.

• **Initialization:** On input $(\mathsf{init}, f, \vec{xid}, p, p')$, from $\mathcal{F}_{verify}$, $\mathcal{S}$ initializes its local copies of $\mathcal{F}_{pre}$, $\mathcal{F}_{rnd}$, $\mathcal{F}_{commit}$, $\mathcal{F}_{transmit}$ as parties in Π_{verify} do. Then $\mathcal{S}$ simulates running $\mathcal{F}_{pre}$ to generate tuples and $\mathcal{F}_{rnd}$ to generate the randomness. For $p(id) \in \mathcal{C}$, it makes $\mathcal{F}_{rnd}$ generate $r(id)$ that $\mathcal{S}$ receives from $\mathcal{F}_{verify}$.

If the execution has not failed, then $\mathcal{A}$ expects that all (valid) tuples and the randomness $r(id)$ for $p(id) \in \mathcal{C}$ are copied to $\mathcal{F}_{commit}$. If the execution fails, $\mathcal{S}$ sends (stop) to $\mathcal{F}_{verify}$.

• **Input Commitment:** On inputs $(\mathsf{commit\_input}, id, x)$ and $(\mathsf{commit\_input}, id)$, $\mathcal{S}$ simulates sending (commit, id, x) and (commit, id) to $\mathcal{F}_{commit}$.

• **Message Commitment:** On inputs $(\mathsf{send\_msg}, id, x)$ and $(\mathsf{send\_msg}, id)$, $\mathcal{S}$ simulates sending $(\mathsf{transmit}, id, x)$ to $\mathcal{F}_{transmit}$. On input $(\mathsf{commit\_msg}, id)$, $\mathcal{S}$ simulates sending $(\mathsf{mcommit}, id, id', x)$, $(\mathsf{mcommit}, id, id', x')$ to $\mathcal{F}_{commit}$ by $P_{p(id)}$, $P_{p'(id)}$ respectively.

If $(id, id', \perp)$, $(\mathsf{cheater}, p(id))$, or $(\mathsf{cheater}, p'(id))$ should be output by $\mathcal{F}_{commit}$, then $\mathcal{S}$ simulates sending (reveal, id) to $\mathcal{F}_{transmit}$. It then writes $comm[id] \leftarrow x$ in its local copy of $\mathcal{F}_{commit}$, where x is the value that was initially transmitted.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$, if $p(id) \in \mathcal{C}$, $\mathcal{S}$ assigns $comm[id] \leftarrow r(id)$, where $r(id)$ has been simulated to $\mathcal{A}$ during the initialization.

• **Verification:** On input (verify, id), $\mathcal{S}$ decomposes $f(id)$ to basic operations $f_1, \ldots, f_N$, and defines the additional identifiers $id_i^{\mathsf{x}_k}$, $id_i^{\mathsf{y}_k}$, $id_i^{\mathsf{z}_k}$ as the honest parties do. For $p(id) \in \mathcal{C}$ it computes all the intermediate values $comm[id_i^{\mathsf{x}_k}]$ and $comm[id_i^{\mathsf{y}_k}]$, and broadcasts the values $\hat{\vec{x}}$ chosen by $\mathcal{A}$. For $p(id) \notin \mathcal{C}$, broadcasting $\hat{\vec{x}}$ is to be simulated by $\mathcal{S}$ as follows (we use case distinction on types of precomputed tuples causing the broadcast):

1. *Bit decomposition of x in $\mathbb{Z}_{2^m}$:* Need to broadcast $\hat{\vec{x}}_i = [c_1, \ldots, c_m]$, where $c_k \in \{0,1\}$ denotes whether the trusted bit b_k is different from the k-th bit of x. Generate $c_k \xleftarrow{\$} \{0,1\}$.
2. *Multiplication of x_1 and x_2 in $\mathbb{Z}_{2^m}$:* Need to broadcast $\hat{\vec{x}}_i = [(x_1 - a) \bmod 2^m, (x_2 - b) \bmod 2^m]$ for the multiplication triple (a, b, c). Generate $\hat{\vec{x}}_i \xleftarrow{\$} \mathbb{Z}_{2^m}^2$.

$\mathcal{S}$ simulates $(\mathsf{broadcast}, (\mathsf{bc}, p(id)), (id_i^{type}, \hat{\vec{x}}_i)_{i \in [N]})$ using $\mathcal{F}_{transmit}$. If the broadcast succeeds and no $(\mathsf{cheater}, p(id))$ should be output, $\mathcal{S}$ writes $pubv[id_i^{type}] \leftarrow \hat{\vec{x}}_i$ for all honest parties. For the trusted bits, it simulates sending the corresponding messages $(id_{i,k}^{\mathsf{bit}} = 1 - id_{i,k}^{\mathsf{bit}})$ to $\mathcal{F}_{commit}$ for $c_k = 1$, as the honest parties do.

The further computation depends on f_i, and S just simulates sending to $\mathcal{F}_{commit}$ the same messages that the honest parties send (linear combinations and truncations). These operations do not involve any interaction between parties, so nothing needs to be simulated to $\mathcal{A}$.

In the end, $\mathcal{S}$ needs to simulate opening to each party the alleged zero vector $\vec{z}$. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ already knows all the values needed to compute $\vec{z}$. If $p(id) \notin \mathcal{C}$, then $\mathcal{S}$ obtains the difference $z = f(\vec{x}) - y$ from $\mathcal{F}_{verify}$. It takes $\vec{z}_{N+1} \leftarrow [z]$, and $\vec{z}_i \leftarrow \vec{0}$ for all other $i \in \{1, \ldots, N\}$.

• **Cheater detection:** At any time, when $\mathcal{F}_{transmit}$ or $\mathcal{F}_{commit}$ should output a message $(\mathsf{cheater}, k)$, $\mathcal{S}$ outputs $(\mathsf{cheater}, k)$ to $\mathcal{F}_{verify}$. $\mathcal{S}$ discards P_k from its local run of Π_{verify}, i.e. assigns $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 4.34: The simulator $\mathcal{S}_{verify}$

After all the broadcasts and subsequent local operations on $\mathcal{F}_{commit}$ (which do not require any interaction) are simulated, $\mathcal{S}$ simulates opening to each party the alleged zero vector $\vec{z}$. If $p(id) \in \mathcal{C}$, then $\mathcal{S}$ already knows all the values needed to compute $\vec{z}$. If $p(id) \notin \mathcal{C}$, then $\mathcal{S}$ obtains only the difference $f(\vec{x}) - y$ from $\mathcal{F}_{verify}$. However, it needs to simulate the alleged zeroes $\vec{z}_i$ of *each* intermediate basic function f_i. Here we use the fact that, if $p(id) \notin \mathcal{C}$, then it has broadcast $\hat{\vec{x}}$ that indeed corresponds to the computation of $f(\vec{x})$. The only non-zero entries of $\vec{z}$ may come due to the mismatches between $f(\vec{x})$ and y, and these differences $f(\vec{x}) - y$ are provided by $\mathcal{F}_{verify}$.

Correctness. The inputs [messages] of $p(id) \notin \mathcal{C}$, the randomness chosen by $\mathcal{F}_{verify}$, and the inputs [messages] of $p(id) \in \mathcal{C}$ chosen by $\mathcal{A}$ are all stored in $\mathcal{F}_{commit}$. In addition, the precomputed tuples are also stored in the same $\mathcal{F}_{commit}$ by definition of $\mathcal{F}_{pre}$. The functionality $\mathcal{F}_{commit}$ may now be used as a black box, doing computation on all these commitments. It remains to prove that, if all these values are committed properly (that is ensured by $\mathcal{F}_{commit}$ on the condition that $(\mathsf{cheater}, p(id))$ is not output for the prover $P_{p(id)}$), then Π_{verify} does verify the computation of $f(id)$ on input (verify, id).

It easy to see that, if $\vec{z}_i = \vec{0}$ for the alleged zeroes produced by the basic function f_i, then f_i has been computed correctly with respect to the committed inputs and outputs on which it was verified, and $\hat{\vec{x}}_i$ has been computed correctly for f_i. The details of verifying each basic function are analogous to the precomputed tuple generation proof of Lemma 4.6, so we do not repeat the proof here. If all f_i have been computed correctly, then so is the composition of f.

Conversely, if $\vec{z} \neq \vec{0}$ in Π_{verify}, it does not immediately imply that $f(\vec{x}) - y = 0$, since the problem might be in the values broadcast by the prover. The parties of Π_{verify} output $(id, 0)$. Here we use the fact that $\mathcal{F}_{verify}$ also outputs $(id, 0)$ to the parties instead of $(\mathsf{cheater}, p(id))$ during the execution of (verify, id). □

4.5.6 The Main Protocol for Verifiable SMC

The protocol Π_{vmpc} implementing $\mathcal{F}_{vmpc}$ is given in Figure 4.35. The protocol is built on top if the functionality $\mathcal{F}_{verify}$ of Section 4.5.5, used to verify the computation of each output of each round, with respect to the committed inputs, messages, and randomness. A party P_k deviating from the protocol rules is detected when $\mathcal{F}_{verify}$ outputs $(id, 0)$ on input (verify, id), where id identifies some output of P_k, as well as when $\mathcal{F}_{verify}$ outputs $(\mathsf{cheater}, k)$ on any other input.

Lemma 4.11. *Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol Π_{vmpc} UC-realizes $\mathcal{F}_{vmpc}$ in $\mathcal{F}_{verify}$-hybrid model.*

Proof. We use the simulator $\mathcal{S} = \mathcal{S}_{vmpc}$ described in Figure 4.36. The simulator runs a local copy of Π_{vmpc}, together with a local copy of $\mathcal{F}_{verify}$.

In Π_{verify}, each party P_i maintains a local array mlc_i of length n, into which it marks the parties that have been detected in violating the protocol rules. Initially, $mlc_i[k] = 0$ for all $k \in [n]$. If P_k has been detected in cheating, P_i writes $mlc_i[k] = 1$. Π_{vmpc} uses $\mathcal{F}_{verify}$ as a subroutine.

• **In the beginning**, Each party P_i gets the message $(\mathsf{circuits}, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ from $\mathcal{Z}$.

1. *Initializing $\mathcal{F}_{verify}$:* Let the n_{ij}^{ℓ} output wires of the circuit C_{ij}^{ℓ} be enumerated. For all $k \in [n_{ij}^{\ell}]$, the value $id \leftarrow (i, j, \ell, k)$ serves as an identifier for $\mathcal{F}_{verify}$. In addition, for each party P_i, there are identifiers (i, x, k) and (i, r, k) for the enumerated inputs and randomness respectively.

   - For each input wire $id \leftarrow (i, \mathsf{x}, k)$ or $id \leftarrow (i, \mathsf{r}, k)$, let $\mathbb{Z}_{2^m}$ be the ring in which the wire is defined. Define $f(id) \leftarrow \mathsf{id}_{\mathbb{Z}_{2^m}}$, $\vec{xid}(id) \leftarrow [id]$, $p(id) = p'(id) = i$.
   - For each output wire $id \leftarrow (i, j, \ell, k)$, define $f(id)$ as a function consisting of basic operations of Section 4.2, computing the k-th coordinate of $\vec{m}_{ij}^{\ell} \leftarrow C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})$ (this is always possible since every gate of C_{ij}^{ℓ} is by definition some basic operation), $\vec{xid}(id)$ the vector of all the identifiers of $\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1}$ that are actually used by C_{ij}^{ℓ}, $p(id) = i$, $p'(id) = j$.

   Each party sends $(\mathsf{init}, f, \vec{xid}, p, p')$ to $\mathcal{F}_{verify}$.
2. *Randomness generation:* For each randomness input wire $id \leftarrow (i, \mathsf{r}, k)$, each party sends $(\mathsf{commit\_rnd}, id)$ to $\mathcal{F}_{verify}$.
3. *Input commitment:* For each input wire $id \leftarrow (i, \mathsf{x}, k)$, P_i sends $(\mathsf{commit\_input}, id, \vec{x}_i)$ to $\mathcal{F}_{verify}$, and each other party sends $(\mathsf{commit\_input}, id)$ to $\mathcal{F}_{verify}$.

• **For each round** $\ell \in [r]$, P_i computes $\vec{m}_{ij}^{\ell} = C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})$ for all $j \in [n]$, and sends $(\mathsf{send\_msg}, (i, j, \ell, k), m_{ijk}^{\ell})$ to $\mathcal{F}_{verify}$ for all $k \in [|\vec{m}_{ij}^{\ell}|]$.

• **After** r **rounds**, each party P_i outputs $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$ to $\mathcal{Z}$. Let $r' = r$ and $mlc_i[k] \leftarrow 0$ for all $k \in [n]$.

Alternatively, **at any time** before outputs are delivered to parties, if a message $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{verify}$, each party P_i writes $mlc_i[k] \leftarrow 1$. In this case the outputs are not sent to $\mathcal{Z}$. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.

• **After** r' **rounds**:

1. Each party sends $(\mathsf{commit\_msg}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for all $i, j \in [n], \ell \in [r'], k \in [n_{ij}^{\ell}]$. If $(\mathsf{cheater}, i)$ comes from $\mathcal{F}_{verify}$, then each party P_j writes $mlc_j[i] \leftarrow 1$, and the verification of P_i is treated as failed, without even running (verify, id).
2. For each output wire identifier $id \leftarrow (i, j, \ell, k)$, each party sends (verify, id) to $\mathcal{F}_{verify}$, getting an answer b from $\mathcal{F}_{verify}$. If $b = 1$, each party P_j writes $mlc_j[i] \leftarrow 0$. Otherwise, it writes $mlc_j[i] \leftarrow 1$.

• **Finally**, each party P_i outputs to $\mathcal{Z}$ the set of parties $\mathcal{B}_i$ such that $mlc_i[k] = 1$ iff $k \in \mathcal{B}_i$.

Figure 4.35: The protocol Π_{vmpc} for verifiable computations

The simulator $\mathcal{S}$ maintains the commitments $comm[id]$ of the identifiers id denoting the circuit wires whose values are known to the corrupted parties.

• **In the beginning**, $\mathcal{S}$ gets all the circuits $(C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r}$ from $\mathcal{F}_{vmpc}$. These are the same circuits that the parties would have obtained from $\mathcal{Z}$ in Π_{vmpc}.

1. *Initializing $\mathcal{F}_{verify}$:* $\mathcal{S}$ simulates the initialization of $\mathcal{F}_{verify}$. For $i \in \mathcal{C}$, it adjusts the randomness generator of $\mathcal{F}_{verify}$ in such a way that it outputs to $\mathcal{A}$ exactly the same randomness $\vec{r}_i$ that $\mathcal{F}_{vmpc}$ chooses for P_i.
2. *Randomness generation:* $\mathcal{S}$ simulates work of $\mathcal{F}_{verify}$ on inputs $(\mathsf{commit\_rnd}, id)$ for each input wire $id \leftarrow (i, \mathsf{r}, k)$. For all $i \in \mathcal{C}$, the committed randomness $r(id)$ is the same that has been output to $\mathcal{A}$ during the initialization of $\mathcal{F}_{verify}$, chosen by $\mathcal{F}_{vmpc}$, and $\mathcal{S}$ writes $comm[id] \leftarrow r(id)$.
3. *Input commitment:* For each input wire $id \leftarrow (i, \mathsf{x}, k)$, $\mathcal{S}$ simulates work of $\mathcal{F}_{verify}$ on inputs $(\mathsf{commit\_input}, id, x_{ik})$ and $(\mathsf{commit\_input}, id)$, where $\vec{x}_i$ is the vector of inputs of the party P_i. For $i \in \mathcal{C}$, the vector $\vec{x}_i^*$ is chosen by $\mathcal{A}$. $\mathcal{S}$ delivers this $\vec{x}_i^*$ to $\mathcal{F}_{vmpc}$, and writes $comm[id] \leftarrow x_{ik}^*$ for all $id \leftarrow (i, \mathsf{x}, k)$.

• **For each round** $\ell \in [r]$, $\mathcal{S}$ needs to simulate computing the messages $\vec{m}_{ij}^{\ell} = C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})$ for all $j \in \mathcal{C}$. If $i \in \mathcal{C}$, then the message $\vec{m}_{ij}^{*\ell}$ is generated by the adversary, and $\mathcal{S}$ delivers it to $\mathcal{F}_{vmpc}$. If $j \in \mathcal{C}$, then the message $\vec{m}_{ij}^{\ell}$ comes from $\mathcal{F}_{vmpc}$, and $\mathcal{S}$ delivers it to $\mathcal{A}$. In all cases, $\mathcal{S}$ simulates sending $(\mathsf{send\_msg}, (i, j, \ell, k), m_{ijk}^{\ell})$ to $\mathcal{F}_{verify}$ for each entry m_{ijk}^{ℓ} of $\vec{m}_{ij}^{\ell}$.

• **After** r **rounds**, each (honest) party P_i should output $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$ to $\mathcal{Z}$. This does not need to be simulated. Let $r' = r$ and $mlc_i[k] \leftarrow 0$ for all $k \in [n]$.
Alternatively, **at any time** before outputs are delivered to parties, if a message $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{verify}$, $\mathcal{S}$ writes $mlc_i[k] \leftarrow 1$ for each honest party P_i. In this case the outputs are not sent to $\mathcal{Z}$. $\mathcal{S}$ defines $\mathcal{B}_0 = \{k \,|(\mathsf{cheater}, k) \text{ has been output}\}$, and sends $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{vmpc}$ to prevent it from outputting the results to $\mathcal{Z}$. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.

• **After** r' **rounds**:

1. $\mathcal{S}$ simulates sending $(\mathsf{commit\_msg}, (i, j, \ell, k))$ to $\mathcal{F}_{verify}$ for all $i, j \in [n]$, $\ell \in [r']$, $k \in [n_{ij}^{\ell}]$. If either $i \in \mathcal{C}$ or $j \in \mathcal{C}$, it writes $comm[(i, j, \ell, k)] \leftarrow m_{ijk}^{\ell}$. If both $i, j \in \mathcal{C}$, then $\mathcal{A}$ chooses $m_{ijk}^{*\ell}$, and $\mathcal{S}$ writes $comm[(i, j, \ell, k)] \leftarrow m_{ijk}^{*\ell}$
2. For each output wire identifier $id \leftarrow (i, j, \ell, k)$, $\mathcal{S}$ simulates sending (verify, id) to $\mathcal{F}_{verify}$. For each $k \in [n]$, $\mathcal{S}$ simulates the output bit b_k of $\mathcal{F}_{verify}$. If $k \in \mathcal{C}$, and $f'(comm[i]_{i \in \vec{x}id}) \neq comm[id]$ for $f' := f(id)$, then $\mathcal{S}$ simulates $\mathcal{F}_{verify}$ outputting $(id, 0)$, and writes $mlc_i[k] \leftarrow 1$ for each honest party P_i. Otherwise, it simulates $\mathcal{F}_{verify}$ outputting 1, and writes $mlc_i[k] \leftarrow 0$. For all $k \notin \mathcal{C}$, it writes $mlc_i[k] \leftarrow 0$.

• **Finally**, $\mathcal{F}_{vmpc}$ outputs to each party P_i the set of parties $\mathcal{B}$ for which $\vec{m}_{ij}^{*\ell} \neq \vec{m}_{ij}^{\ell}$ has been provided by $\mathcal{S}$ at some point before. It now waits for a set of parties $\mathcal{B}_i$ from $\mathcal{S}$, containing the parties that will be additionally blamed by $\mathcal{B}_i$. Let $\mathcal{B}_i' = \{j \,|mlc_i[j] = 1\}$. $\mathcal{S}$ sends to $\mathcal{F}_{vmpc}$ the sets $\mathcal{B}_i = \mathcal{B}_0 \cup \mathcal{B}_i'$, where $\mathcal{B}_0$ is the set defined in the execution phase.

Figure 4.36: The simulator $\mathcal{S}_{vmpc}$ for verifiable computations

Simulatability. The preprocessing phase of $\mathcal{F}_{vmpc}$ and Π_{vmpc} corresponds to their initialization. Getting the randomness $\vec{r}_i$ for $i \in \mathcal{C}$ from $\mathcal{F}_{vmpc}$, $\mathcal{S}$ simulates initialization of $\mathcal{F}_{verify}$. If it outputs $\perp$, then $\mathcal{S}$ delivers (stop) to $\mathcal{F}_{vmpc}$.

For the private input commitments (commit_input), $\mathcal{S}$ only needs to simulate the committed values of the corrupted parties. All of them all known to $\mathcal{S}$, coming either from $\mathcal{F}_{vmpc}$ or $\mathcal{A}$. For the private randomness commitments (commit_rnd), $\mathcal{S}$ uses the randomness that has been simulated in the preprocessing phase. During the execution phase, $\mathcal{S}$ needs to simulate the messages $\vec{m}_{ij}^{\ell}$ that are computed by the honest parties P_i for corrupted parties P_j (send_msg). It gets all such messages from $\mathcal{F}_{vmpc}$.

At the beginning of the verification phase, $\mathcal{S}$ simulates commitments to the messages (commit_msg) that have been transmitted before. It does not need to know any messages for this. $\mathcal{S}$ simulates work of $\mathcal{F}_{verify}$ on inputs (verify, id), for all circuit output identifiers id. It needs to simulate the side-effect of $\mathcal{F}_{verify}$ that outputs the difference between the actual output of $f(\vec{x})$ and the output y to which the prover was committed. All the verifiable functions f of $\mathcal{F}_{verify}$ correspond to the computation of some output of a circuit C_{ij}^{ℓ} with respect to the committed inputs, randomness, and messages. By definition of $\mathcal{F}_{verify}$, unless at least one message $(\mathsf{cheater}, p(id))$ has been output to each honest party (in this case $p(id) \in \mathcal{C}$), all these values are indeed committed as chosen by the party committing to them. Since each honest party has followed the protocol and computed C_{ij}^{ℓ} properly, and all its commitments are valid, the difference $f(\vec{x}) - y$ should be 0 for honest parties, and so it is easy to simulate.

Correctness. We need to prove that $\mathcal{F}_{vmpc}$ outputs exactly the same values as the parties in Π_{vmpc} would. By definition of $\mathcal{F}_{vmpc}$, there are two kinds of outputs:

1. *The computation output* $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$. Let ℓ be any round. We prove by induction that each message $\vec{m}_{ij}^{\ell}$, seen by the adversary, is consistent with $\mathcal{F}_{vmpc}$'s internal state.

    - *Base:* Initially, there are the inputs $\vec{x}_i$ and the randomness $\vec{r}_i$ in the internal state of $\mathcal{F}_{vmpc}$. So far, for $i \notin \mathcal{C}$, $\mathcal{A}$ has no information about $\vec{x}_i$, $\vec{r}_i$, and for $i \in \mathcal{C}$ it expects $\vec{x}_i = \vec{x}_i^*$, $\vec{r}_i = \vec{r}_i^*$, where $\vec{x}_i^*$ is chosen by $\mathcal{A}$ itself, and $\vec{r}_i^*$ is a uniformly distributed value that has been provided by $\mathcal{F}_{verify}$. Exactly these values are delivered by $\mathcal{S}$ to $\mathcal{F}_{vmpc}$, so the state of $\mathcal{F}_{vmpc}$ is consistent with $\mathcal{A}$.
    - *Step:* In the real world, for each $i \notin \mathcal{C}$, $\mathcal{A}$ chooses all the messages m_{ji}^{ℓ} for $j \in \mathcal{C}$ that will be delivered to P_i. By induction hypothesis, the rest of the messages m_{ji}^{ℓ} for $j \notin \mathcal{C}$ and the inputs/randomness $\vec{x}_i$,$\vec{r}_i$ of the inner state of $\mathcal{F}_{vmpc}$ do not contradict with the view of $\mathcal{A}$. In Π_{vmpc}, $\mathcal{A}$ expects that an honest P_i will now compute each message

$\vec{m}^{\ell+1} = C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell})$. In the inner state of $\mathcal{F}_{vmpc}$, the value $\vec{m}^{\ell+1}$ is computed in exactly the same way.

2. *The sets $\mathcal{B}_i$ of blamed parties.* $\mathcal{F}_{vmpc}$ computes all the messages $\vec{m}_{ij}^{\ell}$ and constructs the set $\mathcal{M}$ of parties j for whom $\vec{m}_{ij}^{\ell} \neq \vec{m}_{ij}^{*\ell}$, where $\vec{m}_{ij}^{*\ell}$ is the value provided by $\mathcal{S}$ (that was actually chosen by $\mathcal{A}$). After that, it receives a couple of messages $(\mathsf{blame}, i, \mathcal{B}_i)$ from $\mathcal{S}$, where $\mathcal{B}_i = \mathcal{B}_0 \cup \mathcal{B}_i'$, and $\mathcal{B}_0 = \{k \mid (\mathsf{cheater}, k)$ has come from $\mathcal{F}_{verify}$ in the execution phase $\}$. The ideal functionality $\mathcal{F}_{vmpc}$ expects $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$. First, we prove that $\mathcal{B}_i \subseteq \mathcal{C}$, i.e. no honest party will be blamed.

   (a) For each $j \in \mathcal{B}_0$, a message $(\mathsf{cheater}, j)$ has come from $\mathcal{F}_{verify}$ at some moment. By definition of $\mathcal{F}_{verify}$, no $(\mathsf{cheater}, j)$ can be sent for $j \notin \mathcal{C}$. Hence $j \in \mathcal{C}$.

   (b) For each $j \in \mathcal{B}_i'$, the proof of P_j has not passed the final verification. For $j \notin \mathcal{C}$, S has committed to $\mathcal{F}_{verify}$ exactly those messages that correspond to the computation of f on the committed input, the randomness $\vec{r}_i$, and the incoming messages $\vec{m}_{ij}^{\ell}$. Hence $j \in \mathcal{C}$.

   Secondly, we prove that $\mathcal{M} \subseteq \mathcal{B}_i$, i.e. all deviating parties will be blamed.

   (a) The first component of $\mathcal{M}$ is $\mathcal{B}_0$ for which $\mathcal{S}$ has sent $(\mathsf{stop}, \mathcal{B}_0)$ during the execution phase. The same set $\mathcal{B}_0$ is a component of each $\mathcal{B}_i$.

   (b) The second component $\mathcal{M}'$ of $\mathcal{M}$ are the parties P_i for whom inconsistency of $\vec{m}_{ij}^{\ell}$ happens in $\mathcal{F}_{vmpc}$.
   We show that if $i \notin \mathcal{B}_k$ for all $k \notin \mathcal{C}$, then $i \notin \mathcal{M}'$. Suppose by contrary that there is some $i \in \mathcal{M}'$, $i \notin \mathcal{B}_k$. If $i \notin \mathcal{B}_k$ for all $k \notin \mathcal{C}$, then the proof of P_i had succeeded for every C_{ij}^{ℓ}. For all $i, j \in [n]$, $\ell \in [r']$, i should have come up with the commitments $\vec{x}_i, \vec{r}_i, \vec{m}_{ij}^{\ell}$ such that $\mathcal{F}_{verify}$ outputs 1 on input (verify, id) for each output wire identifier id. By definition of $\mathcal{S}$, the committed $\vec{x}_i$ are chosen by $\mathcal{A}$ before the execution, in the input commitment phase, the randomness $\vec{r}_i$ is coming from the same distribution as the randomness generated by $\mathcal{F}_{vmpc}$, the incoming messages $\vec{m}_{ji}^{\ell}$ are those that are treated by $\mathcal{F}_{vmpc}$ as being sent to P_i by P_j, and the outgoing messages $\vec{m}_{ij}^{\ell}$ are the same that are computed by $\mathcal{F}_{vmpc}$ (the messages moving between two corrupted parties have been chosen by $\mathcal{A}$). Hence $\vec{m}_{ij}^{\ell} = C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})$ for all $i, j \in [n]$, $\ell \in [r']$, so $i \notin \mathcal{M}'$. □

Lemma 4.12. *Let Π_{vmpc} use the implementation of Π_{verify} that is built on top of Π_{pre}, $\Pi_{transmit}$, and Π_{commit}. Let the initial protocol defined by the circuits C_{ij}^{ℓ} have the following parameters (for one prover):*

Table 4.8: Costs of different phases of Π_{vmpc} for one prover in $\mathbb{Z}_{2^m}$

phase	rounds	$\mathcal{F}_{transmit}$ op	#ops	# bits
pre	5	transmit	$n+3nt$	$n \cdot \mathsf{sh}_n \cdot m(\mu(N_b m + 3N_m) + \kappa(m+3))$ $+nt \cdot \mathsf{sh}_n \cdot \lambda$ $+2nt \cdot \mathsf{sh}_n \cdot M_r$
		forward	nt	$nt \cdot \mathsf{sh}_n \cdot M_r$
		broadcast	1	$n(\mu-1)m \cdot (N_b + \mathsf{sh}_n \cdot 2N_m)$ $+n \cdot \mathsf{sh}_n \cdot (\mu-1)m \cdot (N_b m + N_m)$ $+n \cdot \mathsf{sh}_n \cdot m \cdot \kappa(m+3)$ $+n \cdot \mathsf{sh}_n \cdot \lambda$
exec	$1+r$	transmit	$rn+n$	$\mathsf{sh}_n \cdot (n \cdot M_x + M_c)$
post	2	transmit	n^2	$\mathsf{sh}_n \cdot n \cdot M_c$
		forward	n^2	$\mathsf{sh}_n \cdot n \cdot M_c$
		broadcast	1	$N_g \cdot 2m + n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c)$

- *it has r rounds;*
- *its largest ring is $\mathbb{Z}_{2^m}$;*
- *the number of transmitted bits of the protocol is M_c;*
- *the number of input and randomness bits is M_x and M_r respectively;*
- *the number of bit related gates (bit decomposition, ring extension) is N_b;*
- *the number of multiplication gates is N_m;*
- *the number of input and output wires in the circuits (excluding the intermediate wires) is N_w.*

Let λ be the number of bits used for randomness seeds. The resulting protocol may be seen as split into preprocessing, execution, and postprocessing phases, whose complexity upper bounds are given in Table 4.8 for the optimistic case, where the adversary does not attempt to cheat.

In the pessimistic case, where the adversary does attempt to cheat, up to the final verification the number of rounds at most doubles, and the number of communicated bits increases at most $2n$ times. The cost of final verification increases up to $\log(\max(N_w, N_b \cdot m + N_m))$ times.

Proof. Let $N_g := N_b + N_m$. We have taken the numbers of communicated bits from the previously proven lemmata for Π_{verify}. In the optimistic case, $\mathcal{F}_{transmit}$ works in the cheap mode. We show how Table 4.8 is filled.

Preprocessing cost. The total cost $\mathsf{vcost}_{pre}^{N_b,N_m,m}$ of generating precomputed tuples is taken from Lemma 4.7. The total cost $\mathsf{tr}_{\mathsf{sh}_n \cdot M_r}^{\otimes nt} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M_r}^{\otimes nt} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M_r}^{\otimes nt}$ of generating the randomness is taken from Table 4.6. All the randomness for one

prover can be generated in parallel using the same transmissions and forwardings, so M_r moves into the subindex of tr and fwd. Taking the costs of different $\mathcal{F}_{transmit}$ operations from Table 4.1, the number of rounds of $\mathsf{vcost}_{pre}^{N_b,N_m,m}$ is $\max(1,1{+}2){+}\max(2,2,2) = 5$, regardless of the parameters N_b, N_m, m, and it is $1{+}1{+}1 = 3$ for the randomness generation. Since the preprocessed tuples and the randomness can be generated in parallel, we get the total number of $\max(5,3) = 5$ rounds in the cheap mode of $\mathcal{F}_{transmit}$. The total number of called operations is counted by putting together $\mathsf{vcost}_{pre}^{N_b,N_m,m}$ and $\mathsf{tr}_{\mathsf{sh}_n\cdot M_r}^{\otimes nt} \oplus \mathsf{tr}_{\mathsf{sh}_n\cdot M_r}^{\otimes nt} \oplus \mathsf{fwd}_{\mathsf{sh}_n\cdot M_r}^{\otimes nt}$. The number of different $\mathcal{F}_{transmit}$ operations is counted as follows.

- *Transmit:* Each of the n parties receives its shares of initial precomputed tuples as a single message, as it is sufficient for weak opening. The other $3nt$ transmissions come from generating λ and M_r, where the randomness is treated as a single M_r-bit value.

- *Forward:* The randomness is treated as a single M_r-bit value. There are nt forwarding for its shares.

- *Broadcast:* All broadcasts come from weak openings. The shares of initially opened κ tuples are broadcast as a single message for all tuples. Both weak openings of the pairwise verification can be also treated as a single broadcast message for all tuples. Since all these broadcasts are done simultaneously by the prover, all these values can be sent in a single broadcast.

  There are also n weak openings coming from the λ-bit public randomness used by Π_{pre}. Since in Π_{pubrnd} any party may be chosen for the weak opening, we may let the prover do it, so this opening may included into the same broadcast.

Execution cost. Before the execution starts, each input has to be committed. The total cost of input commitment $\mathsf{tr}_{\mathsf{sh}_n\cdot M_x}^{\otimes n}$ is taken from Table 4.6, where all the M_x bits of one prover are committed in parallel, so M_x moves into subindex.

The M_c bits of the original communication are transmitted in r rounds. On each round, up to $n-1$ distinct transmissions may take place for each party, since it may send something to $n-1$ other parties. Treating the final outputs as a part of these M_c bits, we also accept that a party may send values "to itself", so the upper bound is rn.

Postprocessing cost. The verification cost comes from the complexity of executing $\mathcal{F}_{verify}$ on inputs $(\mathsf{commit\_msg}, id)$ and (verify, id). It consists of the following blocks:

- The total cost of mutually committing the messages $\mathsf{tr}_{\mathsf{sh}_n\cdot M_c}^{\otimes n} \oplus \mathsf{fwd}_{\mathsf{sh}_n\cdot M_c}^{\otimes n}$ is taken from Table 4.6, where all the messages can be committed in parallel,

similarly to inputs and randomness. Although M_c bits need to be delivered only to n parties, different messages should be approved by different receivers. This results in n^2 transmissions (delivering each of the n shares to each of the n senders), and all these messages need to be forwarded.

- The total number of hint broadcast bits $N_g \cdot 2m$ of the postprocessing phase is taken from Lemma 4.8. All these bits are broadcast as a single message.
- The total number of alleged zero bits $n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c)$ is taken from Lemma 4.9. Here we assume that all the outputs of the circuits are exactly the communication messages output by the circuits, so we do not introduce M_y. All n shares of the alleged zero vector are broadcast in parallel by the prover, so it can be treated as a single broadcast.

Putting together the hint broadcast and the alleged zero broadcast, we get one broadcast involving $N_g \cdot 2m + n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c)$ bits.

Cheating overhead. In the pessimistic setting, if any party attempts to cheat, $\mathcal{F}_{transmit}$ works in its expensive mode. As can be seen from Table 4.1, the number of rounds at most doubles, and the total communication increases up to $2n$ times. In the final opening, the function reveal of $\mathcal{F}_{transmit}$ is called instead of broadcast. Since we do not want to reveal all the messages that have been transmitted in parallel, the authentication paths of the Merkle tree for simultaneously sent values may need to be sent to each verifier, so that the signature may be checked. This gives a multiplicative overhead $\log M$ where M is the number of distinct elements sharing one signature. Since the total number of wires is N_w, we may assume that there cannot be more than N_w inputs, randomness, or communication elements committed in the same round. The maximum amount of distinct precomputed tuples is $N_b \cdot m + N_m$. Hence the overhead can be at most $\log(\max(N_w, N_b \cdot m + N_m))$. □

4.5.7 Proof of the Main Theorem

We are now ready to prove Theorem 4.1. We take Π_{vmpc} that is built on top of Π_{verify} (which is in turn using Π_{commit}, Π_{rnd}, Π_{pre}, and $\Pi_{transmit}$).

Correctness. For estimating the correctness error, we need to count the total number of messages sent using $\mathcal{F}_{transmit}$, including all the transmitted, forwarded, and broadcast messages. By message, we mean a bitstring that is signed with one signature. For this, we look at the Table 4.8 and sum up the total number of different $\mathcal{F}_{transmit}$ calls. The total number of transmitted and broadcast messages for one prover is

$$\begin{aligned} n + 3nt + nt + 1 + rn + n + n^2 + n^2 + 1 &= 2n^2 + 4nt + (r+2)n + 2 \\ &\leq 6n^2 + (r+2)n \ . \end{aligned}$$

For n provers, the upper bound is $6n^2(n+r+2)$.

By Lemma 4.1, the error of the underlying $\Pi_{transmit}$ is bounded by $6n^2(n+r+2)\cdot\delta$. The other source of error is Π_{pre}. In order to achieve error at most 2^η, by Lemma 4.6 it is sufficient to take $\mu = 1 + \frac{\eta}{\log N_g} \leq \eta$, and

$$\begin{aligned}\kappa &= \max(\{(n^{1/\mu}+1)\eta, n^{1/\mu}+\mu-1\}) \\ &\leq \max(\{(2^{-\eta}+1)\eta, 2^{-\eta}+\eta\}) \approx \eta \ .\end{aligned}$$

We need this bound to estimate the complexity of preprocessing phase.

Security. We have proven that Π_{vmpc} UC-realizes $\mathcal{F}_{vmpc}$ in Lemma 4.11.

Complexity. First, we estimate the complexity of the optimistic setting, where the adversary does not attempt to stop the protocol. We combine the numbers of Table 4.8 with the costs of particular $\mathcal{F}_{transmit}$ operations of Table 4.1. Since the variables N_b, N_m, M_x, M_c, M_r are estimated for the entire computation of all the n parties, and the costs are linear w.r.t. these values, we do not multiply each number by n to scale it to n provers. The only exception is the parameter κ of the preprocessing phase that is upper bounded by $\eta+1$ for each separate proof, and which is not scaled to n parties, differently from μ. Hence we take everywhere $\kappa' := n\kappa$. Let λ' be the number of bits used in a signature.

We still need to multiply the number of used $\mathcal{F}_{transmit}$ operations by n in the pre- and postprocessing phases.

Preprocessing cost. In order to achieve the reported correctness, we took $\mu \leq \eta$ and $\kappa \leq \eta+1$ (so $\kappa' \leq n(\eta+1)$). We use these numbers for finding an upper bound on communication complexity.

- *Transmit:* The total number of bits transmitted per prover is

  $$n\cdot \mathsf{sh}_n \cdot m(\mu(N_b m + 3N_m) + \kappa'(m+3)) + nt\cdot \mathsf{sh}_n \cdot \lambda + 2nt\cdot \mathsf{sh}_n \cdot M_r \ .$$

  Since N_b, N_m are already counted for all n provers, and the same seed λ can be used with different randomness generators, this number is not multiplied by n. The total number of independent transmissions that need a signature is $(n+3nt)$ for each prover. Cheap mode transmission only adds the signature overhead.

  Using the upper bounds for μ and κ' presented above, we get an upper bound of total bit communication for all n provers:

  $$\begin{aligned}\mathsf{nb}^{\mathsf{tr}}_{pre} &= \mathsf{sh}_n \cdot n\eta m(mN_b + 3N_m) \\ &\quad + \mathsf{sh}_n \cdot n^2\eta m(m+3) + \mathsf{sh}_n \cdot (nt\cdot\lambda + 2nt\cdot M_r) \\ &\quad + n(n+3nt)\lambda' \ .\end{aligned}$$

- *Forward:* The total number of bits is $nt \cdot \mathsf{sh}_n \cdot M_r$, and the complexity of forwarding is the same as of transmission. There are nt forwardings for each prover, so the total number of bits is

$$\mathsf{nb}^{\mathsf{fwd}}_{pre} = \mathsf{sh}_n \cdot nt \cdot M_r + n^2 t \cdot \lambda' \ .$$

- *Broadcast:* The total number of bits is $n(\mu-1)m \cdot (N_b + \mathsf{sh}_n \cdot 2N_m) + n \cdot \mathsf{sh}_n \cdot (\mu-1)m \cdot (N_b m + N_m) + n \cdot \mathsf{sh}_n \cdot m \cdot \kappa'(m+3) + n \cdot \mathsf{sh}_n \cdot \lambda$. The realization of broadcast that we use multiplies this number of bits by n^2. Using the same inequalities as for transmission case, and moving N_b deeper into the brackets, we get

$$\begin{aligned} \mathsf{nb}^{\mathsf{bc}}_{pre} \quad = \quad & \mathsf{sh}_n \cdot n^3 \eta m(mN_b + 3N_m) + n^3 \eta m N_b \\ & + \mathsf{sh}_n \cdot n^4 \eta m(m+3) + \mathsf{sh}_n \cdot n^3 \lambda \\ & + n^2 \cdot \lambda' \ . \end{aligned}$$

Summing together $\mathsf{nb}^{\mathsf{tr}}_{pre} + \mathsf{nb}^{\mathsf{fwd}}_{pre} + \mathsf{nb}^{\mathsf{bc}}_{pre}$, putting all the non-leading terms into o, treating λ, λ' as constants, and assuming for simplicity $n \leq \min(N_b, N_m)$ (each party computes at least one bit decomposition and one multiplication gate), and $\lambda \leq \eta$, we get the total number of bits upper bounded by

$$\mathsf{nb}_{pre} = \mathsf{sh}_n \cdot (4n^3 \eta m(N_b m + 3N_m) + 3n^2 M_r) + o(n^3 \eta m N_b) \ .$$

Execution cost. There are $rn + n$ transmissions per party. Since M_x and M_c are already estimated for all n parties, the cost of this phase is $\mathsf{sh}_n \cdot (n \cdot M_x + M_c) + n^2(r+1)\lambda'$. Treating λ' as constant, we may write the total cost simply as

$$\mathsf{nb}_{exec} = \mathsf{sh}_n \cdot (n \cdot M_x + M_c) + o(rn^2) \ .$$

Postprocessing cost. Translating the values of Table 4.8 to actual communication gives us the following costs:

- *Transmit:* $\mathsf{sh}_n \cdot n \cdot M_c + n^2 \lambda$.
- *Forward:* $\mathsf{sh}_n \cdot n \cdot M_c + 2n^2 \lambda$.
- *Broadcast:* $n^2(N_g \cdot 2m + \lambda) + n^2 \cdot (n \cdot \mathsf{sh}_n \cdot (N_g \cdot 2m + M_c) + \lambda')$. Treating λ, λ' as constants, and assuming $n \leq N_g$ (each party computes at least one non-linear gate), we may write it as

$$\mathsf{nb}_{post} = \mathsf{sh}_n \cdot (2n^3 N_g m + n^2 M_c) + o(n^2 N_g m) \ .$$

Cheating overhead. For estimating the numbers of the pessimistic setting, we look at Table 4.1. The number of rounds for each expensive mode operation is twice as large as the same operation in the cheap mode, and the bit communication is up to $2n$ times larger. Another possibility for the adversary to increase the communication is to fail the last weak opening of alleged zeroes and force all the shares committed so far to be revealed. The weak opening may fail either if the prover clearly broadcasts inconsistent messages, or if some verifier complains that the broadcast values were not correct. In both cases, a strong opening pinpoints the party that has caused the weak opening to fail. Hence a covert adversary will not do it anyway. □

Discussion. Treating the number of parties as a constant, we get the following complexities of different phases:

- *Preprocessing:* $O(\eta m(N_b m + N_r) + M_r)$.
- *Execution:* $O(M_x + M_c + r)$.
- *Postprocessing:* $O(N_g m + M_c)$.

For $n = 5$, the constant of O is already quite large due to the exponential nature of share cost sh_n and the quadratic cost of broadcast. However, for $n = 3$, the constant is very small. The gates involving bit decomposition provide additional multiplicative overhead of m, where 2^m is the size of the ring in which the computation takes place. Otherwise, all the overheads are linear. Our verification method becomes very fast for 3-party protocols, especially if we substitute the alleged zero openings with hash exchange, as described in Section 4.3.3.

4.5.8 Another Protocol for Verification

We provide an alternative implementation of $\mathcal{F}_{verify}$ that can be used in the particular case of computation over finite fields. This time, we propose a solution that does not require communication of parties in the preprocessing phase, and where all necessary precomputation can be done by all parties locally, based on the circuit that they compute. The verification is also based on commitments for which we use the same functionality $\mathcal{F}_{commit}$ of Section 4.5.2 (in a finite field, the corresponding protocol Π_{commit} may use Shamir's sharing). The difference comes from the behaviour of Π_{verify} on inputs (verify, id).

For verification, we use linear probabilistically checkable proofs by Ben-Sasson et al. [9] given in Section 2.7. We let the relation $\mathcal{R}_C$ correspond to the circuit C executed by the party whose observance of the protocol is being verified. In this correspondence, $\vec{v}$ is the tuple of all inputs, outputs, and used random values of

that party. The vector $\vec{w}$ extends $\vec{v}$ with the results of all intermediate computations by that party.

Recall that in Section 2.7 the verifier generates 5 challenges $\vec{q}_1, \ldots, \vec{q}_5$ and the state information $\vec{u}$ with length $|\vec{v}| + 2$. Given the query results $a_i = \langle \vec{\pi}, \vec{q}_i \rangle$ for $i \in \{1, \ldots, 5\}$ and the state information $\vec{u} = [u_0, u_1, \ldots, u_{|\vec{v}|+1}]$, the following two checks have to pass:

$$a_1 a_2 - a_3 - a_4 u_{|\vec{v}|+1} = 0, \tag{$*$}$$

$$a_5 - \langle \vec{v}, [u_0, u_1, \ldots, u_{|\vec{v}|}] \rangle = 0. \tag{$**$}$$

Here $(*)$ is used to show the existence of $\vec{w}$, and $(**)$ shows that a certain segment of $\vec{\pi}$ equals $\vec{v}$. For simplicity, let us reorder the entries of $\vec{\pi}$ and write $\vec{\pi} = \vec{p} \| \vec{v}$, where $\vec{p}$ represents all the other entries of $\vec{\pi}$. Let the entries of challenges $\vec{q}_1, \ldots, \vec{q}_5$ be reordered in the same way.

In the original paper [9], it was shown how the verifier can be given access to $\mathcal{V}^{\vec{\pi}}(\cdot)$ without actually getting any information about $\vec{\pi}$. Unfortunately, it requires homomorphic encryption, and the number of encryptions is linear in the size of the circuit. We show that the availability of honest majority allows the proof to be completed without public-key encryptions.

The multiparty setting introduces a further difference from [9]: the vector $\vec{v}$ can no longer be considered public, as it contains the prover's private values. We thus have to strengthen the HVZK requirement in Definition 2.12, making $\vec{v}$ private to the prover. The LPCP constructions of [9] do not satisfy this strengthened HVZK requirement, but their authors show that this requirement would be satisfied if a_5 were not present. In the following, we propose a construction where just the first check $(*)$ is sufficient, so only $a_1, \ldots, a_4$ have to be published. We prove that the second check $(**)$ will be passed automatically.

The following algorithms are implicitly defined by Ben-Sasson et al.:

- $witness(C, \vec{v})$: if $\vec{v}$ corresponds to a valid computation of C, returns a witness $\vec{w}$ such that $(\vec{v}, \vec{w}) \in \mathcal{R}_C$.
- $proof(C, \vec{v}, \vec{w})$: if $(\vec{v}, \vec{w}) \in \mathcal{R}_C$, it constructs a corresponding proof $\vec{p}$.
- $challenge(C, \tau)$: returns $\vec{q}_1, \ldots, \vec{q}_5, \vec{u}$ that correspond to the randomness τ, such that:
  - for any valid proof $\vec{\pi} = \vec{p} \| \vec{v}$, where $\vec{p} = proof(C, \vec{v}, \vec{w})$ for $(\vec{v}, \vec{w}) \in \mathcal{R}_C$, the checks $(*)$ and $(**)$ succeed with probability 1;
  - for any proof $\vec{\pi^*}$ generated without knowing τ, or such $\vec{w}$ that $(\vec{v}, \vec{w}) \in \mathcal{R}_C$, either $(*)$ or $(**)$ fails, except with negligible probability ε.

These algorithms are used by the new implementation $\Pi^{\mathbb{F}}_{verify}$ of $\mathcal{F}_{verify}$. The protocol is given in Figure 4.37. The initialization of $\Pi^{\mathbb{F}}_{verify}$ is similar to the initialization of Π_{verify}, except that $\mathcal{F}_{pre}$ is no longer being used, and $\mathcal{F}_{pubrnd}$ is initialized instead, so that a single public randomness τ could be generated for each verification later.

Differently from Π_{verify} that supports verification of different bit-related computation, the new protocol $\Pi^{\mathbb{F}}_{verify}$ only allows addition and multiplication gates to be verified in a straightforward way. It is sufficient to model any computation, but the overheads may be larger for computations involving many bit decompositions.

It is important that $\Pi^{\mathbb{F}}_{verify}$ implements $\mathcal{F}_{verify}$ on the condition that, for all functions f to be verified, the honest parties always commit $\vec{x}$ and y such that $f(\vec{x}) = y$. Since the verification mechanism is not designed to protect the prover if it is cheating, we can no longer guarantee that if $f(\vec{x}) \neq y$, then only the difference $\vec{z} = f(\vec{x}) - y$ is leaked to the adversary. This assumption is reasonable for verifiable computation, and this is satisfied by the protocol Π_{vmpc} that uses $\mathcal{F}_{verify}$ as a subroutine.

Proposition 4.1. Let $\mathcal{C}$ be the set of corrupted parties. Assuming $|\mathcal{C}| < n/2$, the protocol $\Pi^{\mathbb{F}}_{verify}$ UC-realizes $\mathcal{F}_{verify}$ in $\mathcal{F}_{transmit}$-$\mathcal{F}_{commit}$-$\mathcal{F}_{pubrnd}$-hybrid model.

Proof. We use the simulator $\mathcal{S} = \mathcal{S}^{\mathbb{F}}_{verify}$ described in Figure 4.38. The simulator runs a local copy of $\Pi^{\mathbb{F}}_{verify}$, together with local copies of $\mathcal{F}_{transmit}$, $\mathcal{F}_{commit}$, $\mathcal{F}_{pubrnd}$.

Simulatability. During the initialization and the commitments, the work of $\mathcal{S}$ is analogous to $\mathcal{S}_{verify}$, except that the run of $\mathcal{F}_{pre}$ does not need to be simulated.

On input (verify, id), $\mathcal{S}$ first needs to simulate the commitment of $\vec{p}$. Its value is chosen by $\mathcal{A}$ for $p(id) \in \mathcal{C}$. The commitment can be simulated without knowing $\vec{p}$ for $p(id) \notin \mathcal{C}$, by properties of $\mathcal{F}_{commit}$. Generation and opening of the challenge τ is reduced to $\mathcal{F}_{pubrnd}$. $\mathcal{S}$ generates a uniformly distributed τ, and simulates $\mathcal{F}_{pubrnd}$, opening τ to $\mathcal{A}$. $\mathcal{S}$ uses $\mathcal{Q}$ to generate $q_1, \ldots, q_4$ and $\vec{u}$ based on τ.

Now the values $a_1, \ldots, a_4$ should be output. For $p(id) \in \mathcal{C}$, $\mathcal{S}$ may compute all these values itself, based on the commitments of $P_{p(id)}$ and the challenge τ. For $p(id) \notin \mathcal{C}$, we assume that $f(\vec{x}) = y$, and hence the check $(*)$ should pass. We use the fact that the LPCP that we use is statistical HVZK, and the values $a_s = \sum_k \pi_k \cdot q^s_k$ can be simulated knowing the trapdoor τ. Knowing τ gives enough information about how to generate $a_1, \ldots, a_4$ in such a way that $(*)$ succeeds with probability 1, and the distribution of a_s is the same as for the real proof $\vec{\pi}$. Namely, as shown in [9], $\mathcal{S}$ may generate $a_1, a_2, a_3 \xleftarrow{\$} \mathbb{Z}_q$ due to the randomness $\delta_A, \delta_B, \delta_C$ contained in $\vec{p}$, and then compute $a_4 = (a_1 \cdot a_2 - a_3) \cdot u^{-1}_{|\vec{v}|+1}$, where knowing τ is sufficient for computing $u_{|\vec{v}|+1} = Z_S(\tau)$ for a certain public polynomial Z_S defined in Section 2.7. $\mathcal{S}$ simulates opening of these values.

In $\Pi^{\mathbb{F}}_{verify}$, each party works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ committed to $comm[id]$, the operation $f(id)$ to verify, and the identifiers $\vec{xid}(id)$ of the inputs on which $f(id)$ should be verified w.r.t. the output identified by id. The prover stores the committed values in a local array $comm$. The verifiers store the helpful values published by the verifier in an array $pubv$. The messages that are not committed yet are stored by the sender and the receiver in a local array $sent$. Π_{verify} uses $\mathcal{F}_{transmit}$, $\mathcal{F}_{pubrnd}$, and $\mathcal{F}_{commit}$ as subroutines. Let the computation take place in $\mathbb{Z}_q$ for a prime q.

• **Initialization:** On input $(\mathsf{init}, \hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'})$, where the domains of the mappings $\hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'}$ are all the same, initialize $comm$ and $sent$ to empty arrays. Assign the mappings $f \leftarrow \hat{f}$, $\vec{xid} \leftarrow \hat{\vec{xid}}$, $p \leftarrow \hat{p}$, $p' \leftarrow \hat{p'}$.

Initializing subroutine protocols:

- *Initialize* $\mathcal{F}_{transmit}$: For all $id \in \mathsf{Dom}(f)$ s.t $p(id) \neq p'(id)$, define the mappings s, r, f' such that $s(id) \leftarrow p(id)$, $r(id) = f'(id) \leftarrow p'(id)$. For all $i \in [n]$, define an identifier $id' \leftarrow (\mathsf{bc}, i)$ that will be used for broadcast, and $s(id) \leftarrow i$, $r(id) \leftarrow \perp$, $f'(id) \leftarrow \perp$. Send $(\mathsf{init}, s, r, f')$ to $\mathcal{F}_{transmit}$.
- *Initialize* $\mathcal{F}_{commit}$: For commitments of non-random wires, take $\tilde{p}(id) \leftarrow p(id)$, and $\tilde{m}(id) \leftarrow m$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. If $p(id) \neq p'(id)$, generate a new identifier id' and define additionally $\tilde{m}(id') \leftarrow m(id)$, $\tilde{p}(id') \leftarrow p'(id)$. After doing it for all id, send $(\mathsf{init}, \tilde{m}, \tilde{p})$ to $\mathcal{F}_{commit}$.
- *Initialize* $\mathcal{F}_{rnd}$: For commitments of random wires, take $\tilde{p}(id) \leftarrow p(id)$, and $\tilde{m}(id) \leftarrow m$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. After doing it for all id, send $(\mathsf{init}, \tilde{m}, \tilde{p})$ to $\mathcal{F}_{rnd}$.
- *Initialize* $\mathcal{F}_{pubrnd}$: For a single identifier id_τ, define $m(id_\tau) = \lambda$, where λ is the number of bits in the challenge, that depends on the security parameter. Send (init, m) to $\mathcal{F}_{pubrnd}$.

• **Cheater detection:** At any time, when $\mathcal{F}_{transmit}$, $\mathcal{F}_{pubrnd}$ or $\mathcal{F}_{commit}$ outputs a message $(\mathsf{cheater}, k)$, output $(\mathsf{cheater}, k)$ to $\mathcal{Z}$. Treat P_k as if it has left the protocol, i.e. assign $\mathcal{P} \leftarrow \mathcal{P} \backslash \{k\}$. If $(\mathsf{cheater}, p(id))$ comes from $\mathcal{F}_{commit}$ during executing (verify, id), then all parties output $(id, 0)$ instead of $(\mathsf{cheater}, p(id))$, denoting that the proof of $p(id)$ has failed.

• **Input, Message, Randomness Commitment:** The parties behave exactly in the same way as in Π_{verify}, using $\mathcal{F}_{commit}$ for all commitments. Let $\vec{v}$ be the vector of all committed inputs, randomness, and communication, and let id^v_k be the identifier such that $comm[id^v_k] = v_k$.

• **Verification:** On input (verify, id):

- Let $\vec{v} = \vec{x} \| \vec{r} \| \vec{m}$ be the vector of all committed inputs $\vec{x}$, randomness $\vec{r}$, and messages $\vec{m}$ for $P_{p(id)}$. It computes $\vec{w} \leftarrow witness(f(id), \vec{v})$ and $\vec{p} = proof(f(id), \vec{v}, \vec{w})$. It sends $(\mathsf{commit}, id^p_k, p_k)$ to $\mathcal{F}_{commit}$ for each entry p_k of $\vec{p}$
- After all commitments are done, each party sends $(\mathsf{pubrnd}, id_\tau)$ to $\mathcal{F}_{pubrnd}$, receiving back the challenge τ.
- Each party generates $(\vec{q}^1, \ldots, \vec{q}^5, \vec{u}) = challenge(f(id), \tau)$. Let $id^\pi_1, \ldots, id^\pi_\ell := [id^p_1, \ldots, id^p_{|\vec{p}|}, id^v_1, \ldots, id^v_{|\vec{v}|}]$. Each party sends $(id^a_1 = \sum_{k=1}^{\ell} id^\pi_k \cdot q^1_k)$, $\ldots$, $(id^a_4 = \sum_{k=1}^{\ell} id^\pi_k \cdot q^4_k)$, and then (open, id^a_1),$\ldots$,(open, id^a_4) to $\mathcal{F}_{commit}$, getting back $a_1, \ldots, a_4$.
- Each party checks $a_1 a_2 - a_3 - a_4 u_{|\vec{w}|+1} = 0$. If it holds, output 1 to $\mathcal{Z}$. Otherwise, output 0 to $\mathcal{Z}$.

Figure 4.37: Real protocol $\Pi^{\mathbb{F}}_{verify}$

• **Initialization:** $\mathcal{S}$ gets $(\mathsf{init}, f, \vec{xid}, p, p')$ from $\mathcal{F}_{verify}$. It initializes its local copies of $\mathcal{F}_{pubrnd}$, $\mathcal{F}_{commit}$, $\mathcal{F}_{rnd}$, $\mathcal{F}_{transmit}$ as parties in $\Pi^{\mathbb{F}}_{verify}$ do.

• **Input, Message, Randomness Commitment:** $\mathcal{S}$ simulates the corresponding inputs similarly to $\mathcal{S}_{verify}$.

• **Verification:** On input (verify, id), commitment of $\vec{p}$ needs to be simulated. For $p(id) \notin \mathcal{C}$, it can be done without knowing $\vec{p}$ by properties of $\mathcal{F}_{commit}$. For $p(id) \in \mathcal{C}$, $\mathcal{S}$ computes $\vec{w} \leftarrow witness(f(id), \vec{v})$ and $\vec{p} = proof(f(id), \vec{v}, \vec{w})$, where $\vec{v} = \vec{x} \| \vec{r} \| \vec{m}$ is the vector of all committed inputs $\vec{x}$, randomness $\vec{r}$, and messages $\vec{m}$ for $P_{p(id)}$, all of which have already been simulated to $\mathcal{A}$ during the commitment steps.

$\mathcal{S}$ simulates $\mathcal{F}_{pubrnd}$, resulting in outputting a challenge τ to $\mathcal{A}$. It then generates $(\vec{q^1}, \ldots, \vec{q^5}, \vec{u}) = challenge(f(id), \tau)$.

In the end, $\mathcal{S}$ needs to simulate opening the values $a_1, \ldots, a_4$. For $p(id) \in \mathcal{C}$, $\mathcal{S}$ computes these values from the commitments of $P_{p(id)}$ that are all known to $\mathcal{S}$. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ generates $a_1, a_2, a_3 \xleftarrow{\$} \mathbb{Z}_q$, and $a_4 = (a_1 \cdot a_2 - a_3) \cdot (Z_S(\tau))^{-1}$, where Z_S is a certain public polynomial defined in Section 2.7.

Cheater detection: At any time, when $\mathcal{F}_{transmit}$, $\mathcal{F}_{pubrnd}$, or $\mathcal{F}_{commit}$ should output a message $(\mathsf{cheater}, k)$, $\mathcal{S}$ outputs $(\mathsf{cheater}, k)$ to $\mathcal{F}_{verify}$. $\mathcal{S}$ discards P_k from its local run of $\Pi^{\mathbb{F}}_{verify}$, assigning $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$ and $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$.

Figure 4.38: The simulator $\mathcal{S}^{\mathbb{F}}_{verify}$

Correctness. In the real protocol, the randomness chosen by $\mathcal{F}_{verify}$, and all the inputs and messages of $p(id) \in \mathcal{C}$ (possibly chosen by $\mathcal{A}$) are all stored in $\mathcal{F}_{commit}$. Then $\mathcal{F}_{commit}$ is used as a black box, doing computations on all these commitments. It remains to prove that, assuming that all the inputs $\vec{x} = comm[i]_{i \in \vec{xid}(id)}$ and the outputs $y = comm[id]$ have been committed properly, Π_{verify} does verify the computation of $f(id)$ on input (verify, id).

First, we need to show that, if $f(\vec{x}) = y$, then the proofs succeed for all parties that followed the protocol. For $p(id) \notin \mathcal{C}$, $\mathcal{S}$ has chosen a_i in such a way that the verification always succeeds. If $p(id) \in \mathcal{C}$ does have followed the protocol, then it would take $\vec{\pi} = proof(f(id), \vec{w}, \vec{v})$ where $\vec{w} = witness(f(id), \vec{v})$. Since all honest verifiers have used $challenge(f(id), \tau)$ to generate $\vec{q_1}, \ldots, \vec{q_5}$ and $\vec{u}$ on a truly uniformly distributed τ, the checks $(*)$ and $(**)$ succeed with probability 1. If $p(id)$ has followed the protocol, it would not be claimed a cheater during execution of the subroutines $\mathcal{F}_{commit}$, $\mathcal{F}_{pubrnd}$, $\mathcal{F}_{transmit}$, so passing $(*)$ is sufficient for the proof to succeed.

Conversely, we need to show that if the proof succeeds, then $f(\vec{x}) = y$. For this, we prove the following:

1. If the verification succeeds, the implicit check $(**)$ passes. By definition, this check verifies that the part $\vec{v}$ of the proof $\vec{\pi} = \vec{p} \| \vec{v}$ corresponds to all the inputs and outputs of the circuit defined by $f(id)$. All the commitments were done on approval of all the honest parties inputting $(\mathsf{commit\_input}, id')$,

$(\mathsf{commit\_rnd}, \vec{id'})$, $(\mathsf{send\_msg}, \vec{id'})$, and $(\mathsf{commit\_msg}, \vec{id'})$ for all identifiers $\vec{id'} \in \vec{xid}||[id]$ that are used as a part of $\vec{\pi}$ later. Hence if the commitments succeeded, then $(**)$ would also pass.

2. If the verification succeeds, the explicit check $(*)$ passes. Assuming that all commitments and all openings have succeeded, the value that is finally opened is $a_1 a_2 - a_3 - a_4 u_{|\vec{v}|+1}$, where $a_i = \langle \vec{\pi}, \vec{q_i} \rangle$ for the vectors $\vec{q_i}$ and $\vec{u}$ that have been generated using $challenge(f(id), \tau)$. Hence if this value is 0, then $(*)$ would pass.

By the knowledge property of LPCP, since τ has been opened *after* the prover has been committed to $\vec{\pi}$ that it cannot modify anymore by properties of $\mathcal{F}_{commit}$, and τ indeed comes from random uniform distribution by properties of $\mathcal{F}_{pubrnd}$, any $\vec{\pi^*}$ that satisfies $(*)$ and $(**)$ is a valid proof of existence of a witness $\vec{w}$ such that $(\vec{v}, \vec{w}) \in \mathcal{R}_{f(id)}$. □

Proposition 4.2. Let N_m be the total number of non-linear gates, and N_w the total number of wires in the prover's circuit. Compared to the protocol Π_{verify}, the protocol $\Pi^{\mathbb{F}}_{verify}$ has the following efficiency gains and losses:

- *Offline:* No communication takes place in the preprocessing phase of $\Pi^{\mathbb{F}}_{verify}$, while tuple generation is the bottleneck for Π_{verify}.
- *Online:* $\Pi^{\mathbb{F}}_{verify}$ wins in ca. $\mathsf{bc}^{\otimes n}_{N_m \cdot \log q}$ and loses in ca. $\mathsf{bc}_{N_w \cdot \log q}$.

Proof. It is only possible to directly compare the circuits whose only non-linear gates are the multiplication gates. Let N_m be the total number of non-linear gates, and N_w the total number of wires in the circuit. Let m be the average bit size of the ring in which the computation of Π_{verify} takes place. Let M_y be the total number of output bits. We see which values need to be communicated only in the online phase of Π_{verify}, and which only in the online phase of $\Pi^{\mathbb{F}}_{verify}$.

- In Π_{verify}, according to Table 4.7, $N_m \cdot 2m$ hint bits need to be broadcast, and $n \cdot \mathsf{sh}_n \cdot (N_m \cdot 2m + M_y)$ values need to be broadcast to open the alleged zeroes. The total cost is $\mathsf{bc}_{N_m \cdot 2m} \oplus \mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot (N_m \cdot 2m + M_y)}$.
- In $\Pi^{\mathbb{F}}_{verify}$, the prover needs to commit to $\vec{p}$, resulting in n transmissions of $|\vec{p}| \cdot \log q$ bits. As shown in Section 2.7, $|\vec{p}| = 4 + N_m + N_w$. The parties need to generate the public randomness of complexity $\mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot \lambda} \oplus \mathsf{rev}^{\otimes nt}_{\mathsf{sh}_n \cdot \lambda}$. In order to open a_1,a_2,a_3,a_4, the parties do n broadcasts of $4 \log q$ bits each. The total additional cost is $\mathsf{tr}^{\otimes n}_{\log q \cdot (4 + N_m + N_w)} \oplus \mathsf{tr}^{\otimes nt}_{\mathsf{sh}_n \cdot \lambda} \oplus \mathsf{rev}^{\otimes nt}_{\mathsf{sh}_n \cdot \lambda} \oplus \mathsf{bc}^{\otimes n}_{4 \log q}$.

Assuming for simplicity that $q \approx 2^m$, $\mathsf{sh}_n = 1$ (Shamir's sharing is possible in $\mathbb{Z}_q$) and that n transmissions are approximately as complex as one broadcast,

treating λ as a constant, we get that the main overhead of Π_{verify} comes from the alleged zero broadcasts $\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot (N_m \cdot 2m + M_y)}$, and the main overhead of $\Pi^{\mathbb{F}}_{verify}$ comes from $\mathsf{tr}^{\otimes n}_{N_w \cdot \log q}$, which is approximately $\mathsf{bc}_{N_w \cdot \log q}$. We see that $\Pi^{\mathbb{F}}_{verify}$ is less efficient if there are many linear gates, which are free for Π_{verify} (the complexity of Π_{verify} does not depend on N_w). However, $\Pi^{\mathbb{F}}_{verify}$ does not have such a huge alleged zero overhead. Without taking into account M_y, we get additive advantage $\mathsf{bc}^{\otimes n}_{N_m \cdot \log q}$ and disadvantage $\mathsf{bc}_{N_w \cdot \log q}$ bits for $\Pi^{\mathbb{F}}_{verify}$. □

For $n = 3$, we have proposed a more efficient method for checking alleged zeroes in Section 4.3.3, such that their number becomes much less important. Also, while we are comparing only the operations that require communication, we should also take into account that the generation of $\vec{p}$ by the verifier is done by computing the Fast Fourier Transform [89], which immediately gives a multiplicative overhead $O(\log(N_m + N_w))$ to local computation . Finally, the most important advantage of Π_{verify} is that it naturally supports bit-related operations. Hence we do not claim that one of our protocols has a clearer advantage before the other protocol, and choosing between them depends on the context.

4.6 Extensions

In this section we describe possible optimizations and extensions of the transformation described in Section 4.3. In this undertaking, we are motivated by the Sharemind protocol set [17, 68, 59, 56]. Almost all Sharemind protocols are generated from a clear description of how messages are computed and exchanged between parties [67]. The application itself is described in a high-level language that is compiled into bytecode [15], instructing the Sharemind virtual machine (VM) to call the compiled lower-level protocols in certain order with certain arguments. These protocols call the networking methods in order to send a sequence of values to one of the other two computation servers, or to receive messages from them, thus representing the local computation of each party.

There are over 100 primitive protocols that may be called by the VM, compiled from higher-level descriptions. During compilation, these protocols undergo an intermediate format that is very close to circuits of Section 4.2.

In all Sharemind protocols currently in use, the commitment of randomness can be simplified. Any random value is known by exactly two parties out of three (each pair of parties has a common seed). Hence any random value r used by the prover is already shared in the same manner as the messages, i.e $r = r + 0$. Even the seed does not need to be generated jointly, since the protocols are constructed in such a way that the randomness of P_i and P_j is only needed to hide data from the third party P_k, and choosing it in a bad way does not give any benefits to P_i.

4.6.1 Additional Circuit Operations

The operations of circuits that represent local computation of parties in Sharemind protocols are coming from a certain finite set Op. We have extended the basic set of verifiable circuit operations to cover the set Op, and also added verification of the shuffle protocol that has a different description. This is sufficient to represent all Sharemind protocols that are presented in [17, 68, 59, 56]. We note that Sharemind multiplication protocol (Section 3.1.5) only needs multiplications to be verified, so these extensions are useful, but not essential for verifiability of basic Sharemind.

Comparison. The computation of a shared bit $[\![y]\!]$ from $[\![x_1]\!], [\![x_2]\!] \in \mathbb{Z}_{2^m}$, indicating whether $x_1 < x_2$, proceeds by the following composition. First, convert the inputs to the ring $\mathbb{Z}_{2^{m+1}}$, let the results be $[\![x'_1]\!]$ and $[\![x'_2]\!]$. Next, compute $[\![w]\!] = [\![x'_1]\!] - [\![x'_2]\!]$ in the ring $\mathbb{Z}_{2^{m+1}}$. Finally, decompose $[\![w]\!]$ into bits and let $[\![y]\!]$ be the highest bit.

Integer Division and Remainder. The verification is reduced to the equality $[\![x]\!] = [\![z]\!] \cdot [\![y]\!] + [\![w]\!]$ and the inequality $[\![w]\!] \leq [\![y]\!]$. This represents $z = x/y$ as well as $w = x \bmod y$. The equality needs to be verified in $\mathbb{Z}_{2^{2m+1}}$ to avoid overflows, which needs conversion of x and y to a larger ring. The values $[\![z]\!]$ and $[\![w]\!]$ are committed by the prover, each as m bits over $\mathbb{Z}_{2^{2m+1}}$, using trusted bits.

Bit shifts. To compute $[\![y]\!] = [\![x]\!] \ll [\![x']\!]$, where $[\![y]\!]$ and $[\![x]\!]$ are shared over $\mathbb{Z}_{2^n}$ and $[\![x']\!]$ is shared over $\mathbb{Z}_n$, the parties need a precomputed *characteristic vector* (CV) tuple $([\![r]\!], [\![\vec{s}]\!])$, where $[\![r]\!]$ is shared over $\mathbb{Z}_n$, $[\![s_i]\!]$ are shared over $\mathbb{Z}_{2^n}$, the values s_i are bits, the length of $\vec{s}$ is n, and $s_i = 1$ iff $i = r$. The prover broadcasts $\hat{x} = r - x' \in \mathbb{Z}_n$. The verifiers compute $[\![\vec{s'}]\!] = \mathsf{rot}(\hat{x}, [\![\vec{s}]\!])$, defined by $[\![s'_i]\!] = [\![s_{(i+\hat{x}) \bmod n}]\!]$ for all $i < n$. Note that $s'_i = 1$ iff $i = x'$. The verifiers compute $[\![2^{x'}]\!] = \sum_{i=0}^{n-1} 2^i [\![s'_i]\!]$ and multiply it with $[\![x]\!]$ (using a multiplication triple). They compute the alleged zero $[\![z]\!] = [\![r]\!] - [\![x']\!] - \hat{x}$, as well as two alleged zeroes from the multiplication.

To compute $[\![y]\!] = [\![x]\!] \gg [\![x']\!]$, the parties first reverse $[\![x]\!]$, using bit decomposition. They shift the reversed value left by $[\![x']\!]$ positions, and reverse the result again.

During precomputation phase, the CV tuples have to be generated. Their correctness control follows Section 4.3.2, with the following pairwise verification operation. Given tuples $([\![r]\!], [\![\vec{s}]\!])$ and $([\![r']\!], [\![\vec{s'}]\!])$, the verifiers compute $[\![\hat{r}]\!] = [\![r']\!] - [\![r]\!]$, declassify it, compute $[\![\hat{\vec{s}}]\!] = [\![\vec{s}]\!] - \mathsf{rot}(\hat{r}, [\![\vec{s'}]\!])$, declassify it and check that it is a vector of zeroes. Recall (Section 4.3.2) that we need the pairwise verification to only point out whether one tuple is correct and the other one is not.

Rotation. The computation of $[\![\vec{y}]\!] = \mathsf{rot}([\![x']\!], [\![\vec{x}]\!])$ for $[\![\vec{x}]\!], [\![\vec{y}]\!] \in \mathbb{Z}_{2^n}^m$ and $[\![x']\!] \in \mathbb{Z}_m$ could be built from bit shifts, but a direct computation is more efficient. The parties need a *rotation tuple* $([\![r]\!], [\![\vec{s}]\!], [\![\vec{a}]\!], [\![\vec{b}]\!])$, where $[\![r]\!]$ and $[\![\vec{s}]\!]$ are a CV tuple (with $r \in \mathbb{Z}_m$ and $\vec{s} \in \mathbb{Z}_{2^n}^m$), $\vec{a} \in \mathbb{Z}_{2^n}^m$ is random and the elements of $\vec{b}$ satisfy $b_i = a_{(i+r) \bmod m}$. The prover broadcasts $\hat{r} = x' - r$ and $\hat{\vec{x}} = \vec{x} - \vec{a}$. The verifiers can now compute

$$[\![c_i]\!] = \langle \hat{\vec{x}}, \mathsf{rot}(i, [\![\vec{s}]\!]) \rangle \qquad (i \in \{0, \ldots, m-1\})$$
$$[\![\vec{y}]\!] = \mathsf{rot}(\hat{r}, [\![\vec{c}]\!]) + \mathsf{rot}(\hat{r}, [\![\vec{b}]\!]) \ .$$

Here each c_i is equal to some $\hat{x}_i$. The correctness of the computation follows from $\vec{c} = \mathsf{rot}(r, \hat{\vec{x}})$. The procedure gives the alleged zeroes $[\![z']\!] = [\![x']\!] - [\![r]\!] - \hat{r}$ and $[\![\vec{z}]\!] = [\![\vec{x}]\!] - [\![\vec{a}]\!] - \hat{\vec{x}}$.

The pairwise verification of rotation tuples $\mathbf{T} = ([\![r]\!], [\![\vec{s}]\!], [\![\vec{a}]\!], [\![\vec{b}]\!])$ and $\mathbf{T}' = ([\![r']\!], [\![\vec{s'}]\!], [\![\vec{a'}]\!], [\![\vec{b'}]\!])$ works similarly, using the tuple $\mathbf{T}'$ to rotate $[\![\vec{a}]\!]$ by $[\![r]\!]$ positions and checking that the result is equal to $[\![\vec{b}]\!]$ (i.e. subtract one from another, open and check that the outcome is a vector of zeroes). Additionally, pairwise verification of CV tuples is performed on $([\![r]\!], [\![\vec{s}]\!])$ and $([\![r']\!], [\![\vec{s'}]\!])$.

Shuffle. The parties want to apply a permutation σ to a vector $[\![\vec{x}]\!] \in \mathbb{Z}_n^m$, obtaining $[\![\vec{y}]\!]$ satisfying $y_i = x_{\sigma(i)}$. Here $\sigma \in S_m$ is known to the prover and to exactly one of the verifiers [68]. To protect prover's privacy, it must not become known to the other verifier. In the following, we write $[\sigma]$ to denote that σ is known to the prover and to one of the verifiers (w.l.o.g., to V_1).

The parties need a precomputed *permutation triple* $([\rho], [\![\vec{a}]\!], [\![\vec{b}]\!])$, where $\rho \in S_m$, $\vec{a}, \vec{b} \in \mathbb{Z}_n^m$ and $\vec{b} = \rho(\vec{a})$. Both the prover and verifier V_1 sign and send $\tau = \sigma \circ \rho^{-1}$ to V_2 (one of them may send $H(\tau)$; verifier V_2 complains if received τ-s are different). The prover broadcasts $\hat{\vec{x}} = \vec{x} - \vec{a}$. The verifiers compute their shares $(\vec{y}_1, \vec{y}_2)$ of $[\![\vec{y}]\!]$ as $\vec{y}_1 = \tau(\vec{b}_1 + \rho(\hat{\vec{x}}))$ and $\vec{y}_2 = \tau(\vec{b}_2)$, where $\vec{b}_i$ is the i-th verifier's share of $[\![\vec{b}]\!]$. The alleged zeroes $[\![\vec{z}]\!] = [\![\vec{x}]\!] - [\![\vec{a}]\!] - \hat{\vec{x}}$ are produced.

The pairwise verification of triples $([\rho], [\![\vec{a}]\!], [\![\vec{b}]\!])$ and $([\rho'], [\![\vec{a'}]\!], [\![\vec{b'}]\!])$ again works similarly, using the second tuple to apply $[\rho]$ to $[\![\vec{a'}]\!]$. The result is then checked for its equality to $[\![\vec{b'}]\!]$.

4.6.2 Reducing the Number of Bit Decompositions

If we implement our method in a straightforward way, verifying the circuit gate by gate, we need a bit decomposition for every bd and zext gate. Obviously, if some variable x participates in both decompositions, then its bits can be reused, and it does not need to be decomposed twice. In some cases, it is not so obvious, and the circuit should formally be restructured to avoid excessive bit decompositions.

For example, if we already have a bit decomposition for x, and $y = x \cdot 2$ has been computed, then y inherits the bit decomposition without additional need of trusted bits, since $x \cdot 2$ can be seen as a bit operation $x \ll 1$. Since a circuit represents *local* computation of the prover, the circuit can be modified without affecting the execution phase. We give a list of optimizations that we have used in our benchmarks.

Trivial optimizations We make use of standard circuit optimization related to constant propagation and folding. This eliminates the need of precomputed tuples for computing the values that are public.

Inherited Bit Decompositions For each variable x, we mark two flags, whether it needs the binary representation (the bit tuple $(x_1, \ldots, x_m)$, where x_i shared in $\mathbb{Z}_2$), and whether it needs the linear representation (x is shared in $\mathbb{Z}_{2^m}$). It is possible that only one of these representations is needed. Starting from the inputs, we apply bit decompositions on demand, propagating the available bits from gate inputs to gate outputs whenever possible. For example, bitwise operations propagate binary representations, linear operations propagate linear representations, transition to a smaller ring propagates both.

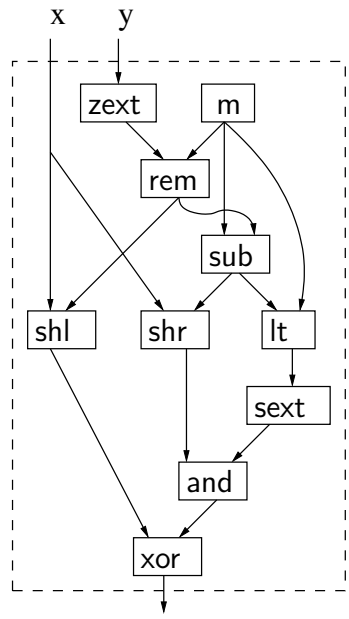


Figure 4.39: Rotation subcircuit

After such propagations, we may still have excessive bit decompositions. Given $x \in \mathbb{Z}_{2^m}$ or $(x_1, \ldots, x_m) \in \mathbb{Z}_2^m$, we want to check if the decomposition $(x'_1, \ldots, x'_m) \in \mathbb{Z}_{2^m}^m$ is necessary. For this, we need look through all the operations that use x or any of x_i as arguments, and see which representation is actually required by these operations. It may happen that these operations, or even subsets of operations, need to be rewritten, so that the same value would be computed using a different representation. In order to do it, we look for subcircuits of certain structure and rewrite them if necessary. In particular, in Sharemind protocols we had two main cases repeating throughout the protocols:

- *Linearizable bitwise operations:* Suppose that the input $x \in \mathbb{Z}_{2^m}$ is first decomposed to the bits $(x_1, \ldots, x_m)$, then $(y_1, \ldots, y_m) \leftarrow op(x_1, \ldots, x_m)$ is computed for a bitwise operation op (possibly represented as m distinct gates for each bit), each bit y_i is converted back to $\mathbb{Z}_{2^m}$, and finally $y = \sum_{i=1}^{m} 2^{i-1} y_i$ is output. Even if bit inheritance allows to compute $y_i \in \mathbb{Z}_{2^m}$

directly from $(x_1, \ldots, x_m) \in \mathbb{Z}_{2^m}^m$ for free, we would still need m trusted bits for the bit decomposition of x. However, depending on op, unless the bits x_i and y_i are not used anywhere else, it is often possible to compute y directly from x without any bit decompositions. We are able to handle at least the following cases (let c be a constant):

$$\begin{aligned} y_i = \neg x_i \quad &\iff \quad y = (2^m - 1 - x) \ , \\ y_i \xleftarrow{\$} \mathbb{Z}_2 \quad &\iff \quad y \xleftarrow{\$} \mathbb{Z}_{2^m} \ , \\ y_i = x_{i+c} \ (y = x \ll c) \quad &\iff \quad y = x \cdot 2^c \ . \end{aligned}$$

- *Choice:* `if` $b = 1$ `then` x `else` y, where $b \in \mathbb{Z}_2$, $x, y, z \in \mathbb{Z}_{2^m}$. This can be computed either bitwise as $z_i = b \wedge x_i + (1 - b) \wedge y_i$, or $z = b \cdot x + (1 - b) \cdot y$ (the latter case requires $b \in \mathbb{Z}_{2^m}$). Depending on the context, even if b needs a bit decomposition to get into $\mathbb{Z}_{2^m}$, it may be more efficient to use $z = b \cdot x + (1-b) \cdot y$ if the variables x, y, z need only the linear representation. In Sharemind protocols, only the binary representation of choices was used.

- *Rotation:* $z = \mathsf{rotate}(x, r)$ is a basic operation of our circuit, which can be verified using a single rotation tuple. In Sharemind protocols, this operation is expressed as a subcircuit involving two bit shifts by private values and a division remainder, which would induce a larger overhead. This subcircuit is depicted in Figure 4.39, where shl and shr are the left and right bit shifts respectively, sub is the subtraction, rem is the division remainder, lt is the comparison ("less than"), zext is the transition to a larger ring, and sext is another transition to a larger ring, turning the bit 0 to the bits $(0, 0, \ldots, 0)$ and the bit 1 to $(1, 1, \ldots, 1)$. The gate m represents the constant m.

Distributive multiplications We rewrite multiplications of the form $x_1 \cdot y_1 + x_1 \cdot y_2$ to $x_1 \cdot (y_1 + y_2)$, reducing the total number of multiplications. This optimization is double-edged, as it may in turn harm some other optimizations. It is better to apply it in the end, after all the other optimizations.

4.6.3 Input and Output Parties

In real applications of sharing based SMC, the parties that provide the inputs and receive the outputs are in general different from the computing parties (see Section 2.2.1). The input [resp. output] parties want to be sure that the verification has been run on the provided inputs that they provided [resp. the received outputs].

Input commitments are handled similarly to messages. First, the input x_j for P_j is shared to x_j^k by the input party P_I that provided x_j, and each share is signed

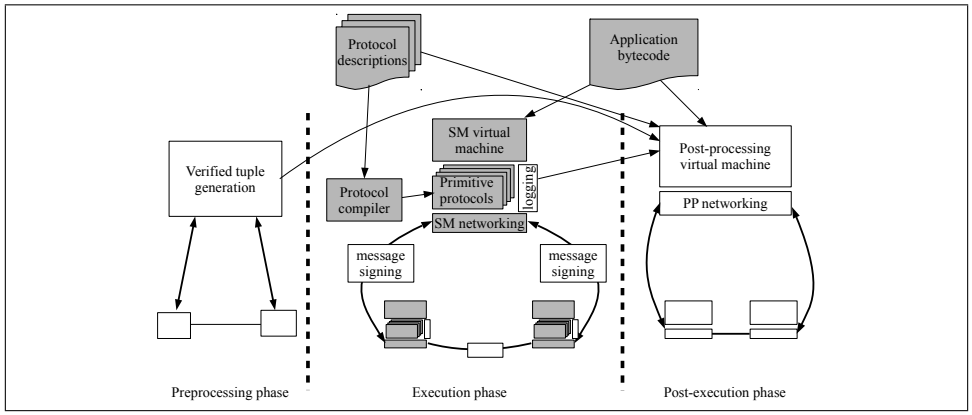


Figure 4.40: Components of Sharemind with verification

by P_I. All shares $(x_j^k)_{k\in[n]}$ and their signatures are sent to P_j that verifies if the sharing is valid, and then forwards x_j^k to P_k. If P_I does not provide valid input, then the computing parties act as if they had not received anything from P_I.

In general setting, P_j sends y_j to the output party P_O directly. We now let also P_k send y_j^k to P_O. In the case of three parties, it is sufficient that both y_j^k are signed by P_j. In this way, P_O should prefer to reconstruct y_j from y_j^k, but if the delivery of some y_j^k fails due to dishonesty of P_k, then it takes y_j. We omit the case of $n > 3$ parties here, since the behaviour of parties becomes less trivial.

4.6.4 Auditability

If a party P has deviated from the protocol, then all honest parties will learn its identity during the post-execution phase. In this case, assuming that P does not drop out from the verification process at all, the honest parties are going to have a set of statements signed by P, pertaining to the values of various messages during all phases, from which the contradiction can be derived. These statements may be presented to a judge that is trusted to preserve the privacy of honest parties.

4.7 Evaluation

4.7.1 Implementation

We have implemented the verification of computations for the Sharemind protocol set. The previously existing (gray) and newly implemented (white) components are depicted in Figure 4.40. We now describe how each phase is implemented.

Preprocessing phase. The verified tuple generator has been implemented in C, compiled with `gcc` ver. 4.8.4, using `-O3` optimization level, and linking against the cryptographic library of OpenSSL 1.0.1k. We have tried to simplify the communication pattern of the tuple generator as much as possible, believing it to maximize performance. On the other hand, we have not tried to parallelize the generator, neither its computation, nor the interplay of computation and communication. Hence we believe that further optimizations are possible.

The generator works as follows. If the parties want to produce u verified tuples, then (i) they select μ and κ appropriately for the desired security level (some particular numbers are given in Section 4.3.2); (ii) the prover sends shares of $(\mu u + \kappa)$ tuples to verifiers; (iii) verifiers agree on a random seed (used to determine, which tuples are opened and which are grouped together) and send it back to the prover; (iv) prover sends to the verifiers κ tuples that were to be opened, as well as the differences between components of tuples that are needed for pairwise verification; (v) verifiers check the well-formedness of opened tuples and check the alleged zeroes stating that they received from the prover the same values, these values match the tuples, and the pairwise checks go through. Steps (ii) and (iv) are communication intensive. In step (iii), each verifier generates a short random vector and sends it to both the prover and the other verifier. The concatenation of these vectors is used as the random seed for step (iv). Step (v) involves the verifiers comparing that they've computed the same hash value. We use SHA-256 as the hash function. After the tuples have been generated, the prover sends to each verifier a signature on the shares that the verifier holds.

To reduce the communication in step (ii) above, we let the prover share a common random seed with each of the verifiers. In this manner, the random values do not have to be sent. E.g. for a multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$, both shares of $\llbracket a \rrbracket$, both shares of $\llbracket b \rrbracket$ and one share of $\llbracket c \rrbracket$ are random. The prover only has to send one of the shares of $\llbracket c \rrbracket$ to one of the verifiers.

Execution phase. This phase is entirely delegated to Sharemind computation servers. During the execution, the virtual machine (VM) reads the description of the privacy-preserving application and executes a certain set of compiled low-level passively secure protocols (see Section 4.6). Sharing the initial inputs among the two remaining parties can be treated as the first step of the privacy-preserving application, so we do not need to implement it separately.

In order to support verification, each computation server of Sharemind must log the randomness it is using, as well as the messages that it has sent or received. Using these logs together with the descriptions of the privacy-preserving application and the primitive protocols, it is possible to restore the execution of the server. As discussed in Section 4.6, since all randomness is known to at least one verifier, no

special resharing of randomness is needed.

We have modified the network layer of Sharemind, making it sign each message it sends, and verify the signature of each message it receives. We have not added the logic to detect whether two outgoing messages belong to the same round or not (in the former case, they could be signed together), but this would not have been necessary, because our compiled protocols produce only a single message for each round. We have used GNU Nettle for the cryptographic operations. For signing, we use 2 Kbit RSA and SHA-256. Beside message signing and verification, we have also added the logging of all outgoing and incoming messages.

Verification phase. The virtual machine of the post-execution phase reads the application bytecode and the log of messages. This is enough to learn which protocols were invoked in which order and with which data during the execution phase. As discussed in Section 4.3.3, since each message is known to at least one verifier, no special resharing of messages is needed.

The information about invoked protocols is present in both the prover's log, as well as in the verifiers' logs. Indeed, the identity of invoked protocols depends only on the application, and on the public data it operates on. This is identical for all computation servers. The post-execution VM then reads the descriptions of protocols and performs the steps described in Section 4.3.3. The post-execution VM has been implemented in Java, translated with the OpenJDK 6 compiler and run in the OpenJDK 7 runtime environment. The verification phase requires parties to sign their messages, we have used 2 Kbit RSA with SHA-256 for that purpose.

4.7.2 The Total Cost of Covertly Secure Protocols

For benchmarking, we have chosen the most general protocols of Sharemind over the ring $\mathbb{Z}_{2^{32}}$: multiplication (MULT32), 128-bit AES (AES128), bitwise conjunction (AND32), conversion from additive sharing (i.e. over $\mathbb{Z}_{2^{32}}$) to xor-sharing (i.e. additive over $\mathbb{Z}_2^{32}$) (A2X32) and vice versa (X2A32). We have measured the total cost of covert security of these protocols, using the tools that we have implemented. Our tests make use of three $2\times$ Intel Xeon E5-2640 v3 2.6 GHz/8GT/20M servers, with 125GB RAM running on a 1Gbps LAN, similarly to the benchmarks reported in Section 3.1 of Chapter 3. Depending on the execution time of a single protocol, we run 10^5, 10^6, or 10^7 protocol instances in parallel, and report the amortized execution time for a single protocol.

Preprocessing. In the described set-up, we are able to generate 100 million verification triples for 32-bit multiplication in ca. 236 seconds (Table 4.9). To verify a single multiplication protocol, we need 6 such triples: we use Sharemind

Table 4.9: Time to generate $u = 10^8$ verified tuples for $\eta = 80$ ($\mu = 4$, $\kappa = 15000$)

tuple	**width**	**prover time**	**verifier time**
Multiplication triples	32 bits	212 s	236 s
	64 bits	309 s	352 s
Trusted bits	32 bits	65 s	72 s
	64 bits	90 s	101 s
xor-shared AND triples	32 bits	212 s	236 s

protocol given in Section 3.1.5 that formally has 3 multiplications per party, but all of them are of the form $x_1 \cdot y_1 + x_1 \cdot y_2 + x_2 \cdot y_1$ and can be trivially rewritten to $x_1 \cdot (y_1 + y_2) + x_2 \cdot y_1$. Hence the amortized preprocessing effort to verify a single 32-bit multiplication is ca. 14 μs. The cost is similar for a single 32-bit AND.

Sharemind uses 6400 AND gates per AES128 block. Each AND gate is just a multiplication, and it requires 6 one-bit triples. The time of generating 10^8 xor-shared 32-bit AND triples is 236 s, and one 32-bit AND triple is the same as 32 ordinary one-bit AND triples (a, b, c) s.t $a \wedge b = c$. Hence the amortized preprocessing effort to verify a single 128-bit AES block is ca. 2.8 ms.

The A2X protocol requires 96 xor-shared AND triples and 64 trusted bits, all of bit width 32. The cost of generating 10^8 trusted bits is 72 s. The amortized preprocessing effort of this protocols is ca. 273 μs. The X2A protocol requires 64 additively shared 32-bit multiplication triples, and 96 trusted bits, all of bit width 32. The amortized effort of this protocol is ca. 220 μs.

Execution. We have measured runtimes of passively secure Sharemind with and without signing and logging. The execution times in milliseconds are given in Table 4.10. If a large number (10^5,10^6,10^7) of these operations are computed in parallel, the amortized time, including all necessary signing and logging, is ca 0.16 μs for AND32 and MULT32, 0.04 ms for one AES128 block, 2.3 μs for A2X, and 5.1 μs for X2A. In general, for sufficiently large inputs, the signing and logging appears to reduce the performance of the current implementation of Sharemind up to three times. It is likely that a more careful parallelization of the networking layer of Sharemind would eliminate most of that overhead.

Verification. Assuming that all the inputs and the communication have been committed, and the preprocessed tuples generated, we run the verification phase in parallel for all 3 provers, and measure the total execution time (for asymmetric protocols, we report the times of all 3 provers). We consider the optimistic setting, where the prover only signs the broadcast message, and the verifiers exchange the

Table 4.10: Times of the execution phase with / without signing and logging (ms)

runs	AND32		MULT32		AES128		A2X32		X2A32	
	w/o	w/	w/o	w/	w/o	w/	w/o	w/	w/o	w/
10^1	0.362	4.75	0.349	3.96	11.3	485	0.785	38.8	0.19	8.75
10^2	0.345	4.42	0.237	3.84	13.4	496	0.928	38.7	1.05	8.59
10^3	0.147	4.58	0.282	4.04	33.0	600	1.73	45.0	2.28	12.8
10^4	0.668	6.37	0.733	5.40	214	726	8.44	55.6	27.3	60.4
10^5	7.46	15.1	8.13	15.1	2090	3740	98.4	227	252	481
10^6	73.9	166	73.8	184	–	–	909	2290	2690	5050
10^7	683	1550	717	1630	–	–	–	–	–	–

Table 4.11: Running times of the verification phase

runs	time (s)								
	AND32	MULT32	AES128	A2X32			X2A32		
				P1	P2	P3	P1	P2	P3
10^1	0.315	0.322	0.472	0.324	0.337	0.323	0.333	0.340	0.337
10^2	0.335	0.337	0.694	0.377	0.387	0.383	0.383	0.413	0.411
10^3	0.387	0.384	1.21	0.496	0.494	0.488	0.465	0.532	0.559
10^4	0.564	0.557	4.46	0.896	0.949	0.930	0.868	1.17	1.21
10^5	0.939	0.952	29.1	2.72	3.08	3.02	2.60	5.31	5.95
10^6	2.72	2.68	–	18.5	21.8	21.4	16.9	37.6	43.0
10^7	16.7	16.7	–	–	–	–	–	–	–

hash of the message to ensure that they got the same message. The results are given in Table 4.11. When performing a large number (10^5,10^6,10^7) verifications in parallel, the cost of verification is ca. 1.7 μs for MULT32 (or AND32), 290 μs for a single AES128 block, 22 μs for A2X32, and 43 μs for X2A32.

Total Cost. When adding the costs of three phases, we find that the total amortized cost of performing a 32-bit multiplication in our three-party SMC protocol tolerating one covertly corrupted party is ca. 16 μs. For a single AND gate, we get 0.5 μs. The total cost of evaluating a 128-bit AES block is ca 3.1 ms. The total cost of conversions between additive and bitwise sharing is ca. 297 μs for A2X32 and 268 μs for X2A32. The results given in Table 4.12.

Instead of Sharemind multiplication, we could apply our verification to the protocol of [3] described in Section 3.1.4. This would reduce the execution phase time, but complexities of the preprocessing and the verification phases remain the

Table 4.12: Total amortized cost of covertly secure protocols

	AND	MULT32	AES128	A2X32	X2A32
cost (μ s)	0.5	16	3100	297	268

same. It is not clear how well that protocol could be integrated with the other Sharemind protocols, so it becomes more interesting when more various protocols composable with [3] will be developed.

From covert to active security. A covert adversary deviates from the protocol only if the chance of being caught is negligible (see Section 2.2.2). In our case, the probability of *not* being caught is negligible, which is stronger that required by the definition of covert adversary. Applying the verification after each round would result in an actively secure protocol.

Although we do not verify each round of the benchmarked protocols, they turn out to be nevertheless *actively* secure. Namely, the protocol set of Sharemind is *private* against active adversaries, as long as no values are declassified [86]. This means that an active adversary is able to break the correctness of the computation, but it does not leak to him any private information unless the results are declassified. In this setting, if declassification is applied only to computation results at the end of the protocol, then prepending it with our verification step gives us an actively secure protocol [66]. The benchmarks may be even better if several protocols are applied sequentially, since only the final outputs need to be verified, while still having active security. Hence we may compare ourselves with the state-of-the-art actively secure protocols.

4.7.3 State-of-the-art Complexity of Actively Secure Integer Multiplication and AES

Let us review the state of the art in performing integer multiplications and AES128 encryptions in actively secure computation protocol sets. All times reported below are amortized over the parallel execution of many protocol instances. All reported tests have used modern (at the time of the test) servers (one per party), connected to each other over a local-area network.

Such protocol sets are based either on garbled circuits or secret sharing (over various fields). Lindell and Riva [71] have recently measured the performance of maliciously secure garbled circuits using state-of-the-art optimizations. Their total execution time for a single AES circuit is around 80ms, when doing 1024 executions in parallel and using the security parameter $\eta = 80$ (bits). The size of their AES circuit is 6800 non-XOR gates. According to [37], a 32-bit multiplier can be built with ca. 1700 non-XOR gates. Hence we extrapolate that such multiplication may take ca. 20ms under the same conditions. Our extrapolation cannot be very precise due to the very different shape of the circuits computing AES or multiplication, but it should be valid at least as an order-of-magnitude approximation.

A protocol based on secret sharing over $\mathbb{Z}_2$ [83] would use the same circuit to perform integer multiplication. In [40], a single non-XOR gate is estimated to require ca. 70 μs during preprocessing (with two parties). Hence a whole 32-bit multiplier would require ca. 120ms. As the preprocessing takes the lion's share of the total costs, there is no need for us to estimate the performance of the online phase.

Recent estimations of the costs of somewhat homomorphic encryption based preprocessing for maliciously secure multiparty computation protocols based on additively secret sharing over $\mathbb{Z}_p$ are hard to come by. In [30], the time to produce a multiplication triple for $p \approx 2^{64}$ is estimated as 2ms for covert security and 6ms for fully malicious security (with two parties, with $\eta = 40$). We presume that the cost is smaller for smaller p, but for $p \approx 2^{32}$, it should not be more than twice as fast. On the other hand, the increase of η to 80 would double the costs [30]. In [31], the time to produce a multiplication triple for $p \approx 2^{32}$ is measured to be 1.4ms (two parties, $\eta = 40$, escape probability of a cheating adversary bounded by 20%).

The running time for actively secure multiplication protocol for 32-bit numbers shared using Shamir's sharing has been reported as 9ms in [28] (with four parties, tolerating a single malicious party). We are not aware of any more modern investigations into Shamir's secret sharing based SMC.

A more efficient N-bit multiplication circuit is proposed in [34], making use computations in $\mathbb{Z}_2$ and in $\mathbb{Z}_p$ for $p \approx N$. Using this circuit instead of the one reported in [37] might improve the running times of certain integer multiplication protocols. Unfortunately, the cost of obtaining multiplication triples for $\mathbb{Z}_p$ is unclear.

In this thesis, we presented a set of protocols that is capable of performing a 32-bit integer multiplication with covert security (on a 1Gbps LAN, with three parties, tolerating a single actively corrupted party, $\eta = 80$, negligible escape probability for a cheating adversary) in 16 μs. This is around two orders of magnitude faster than the performance reported above.

In concurrent work [54], the oblivious transfer methods of [40] have been extended to construct SPDZ multiplication triples over $\mathbb{Z}_p$. They report amortized timings of ca. 200 μs for a single triple with two parties on a 1Gbps network, where $p \approx 2^{128}$ and $\eta = 64$. Reducing the size of integers would probably also reduce the timings, perhaps even bringing them to the same order of magnitude with our results. However, their techniques (as well as most others described here) only work for finite fields, not rings. For fields, there exist methods to reduce the number of discarded triples during triple verification, which also apply for us.

Recently [32], amortized time $0.5\mu s$ was reported for computing a single AES block. However, it takes into account only the online phase. The authors do

not provide benchmarks for preprocessing, but they estimate that using recent mechanisms for doing preprocessing, up to 10^5 AND gates could be computed per second. Assuming that one AES block contains ca 6400 AND gates (as in our benchmarks), this would suffice for around 16 AES blocks per second, or $63ms$ per AES block. In this thesis, we compute a 128-bit AES block with covert security in $3.1ms$, including the preprocessing, which is an order of magnitude faster.

Section 3.1.4 describes the three-party protocol of [3] extended with the verification phase [41]. Their paper does not report the running times, but uses total number of communicated bits per AND gate instead. Their reported number is 30 bits per AND gate for 3 parties. Using the same security parameter $\eta = 40$ (taking $m = 3$), and making use of shared randomness, we get that the generation one 1-bit multiplication triple requires 1 bit of communication and each pairwise verification 4 bits (opening the 2 masked values by 2 verifiers to each other), adding up to $1 + 4 \cdot (m - 1) = 9$ bits for a single verified triple. Since we require a triple for each of the 6 local multiplications of Sharemind protocol, we already get 54 bits. The execution phase requires 6 bits of communication, and the verification phase 24 bits (8 for each party). This is 84 bits in total, or almost three times more. Some additional overhead may come from signatures (their cost becomes negligible as the communication grows). However, our security property is stronger, allowing to pinpoint the cheating party and make the protocol aborting identifiable. Our method is also more generic and allows to easily generate precomputed tuples other than multiplication triples, that are very useful in verifying protocols other than multiplication.

Another work that reports the number of communicated bits is [80]. This is essentially a garbled circuit computation for three parties tolerating one corrupted party. Their reported number of bits is 1504 Kbytes, which is ca 12000 Kbits. Using a similar AES block consisting of 7200 AND gates, and assuming the multiplicative overhead of 84 as discussed above, our solution has 605 Kbits of communication. For $\eta = 80$, which is in any case a sufficiently large security parameter, we get 156 bits per AND gate (derived from Table 4.13), which is 907 Kbit of total communication.

In would be interesting to compare also X2A and A2X with existing solutions, but we could not find similar benchmarks for these protocols.

4.7.4 Estimating the Cost of other Sharemind Protocols

Our implementations of the preprocessing and verification phases are still preliminary, at least compared to the existing Sharemind platform and the engineering effort that has been gone into it. We believe that significant improvements in their running times are possible, even without changing the underlying algorithms or invoking extra protocol-level optimizations. Hence we are looking for another

metric that may predict the running time of the new phases once they have been optimized. Due to the very simple communication pattern of that phase, consisting of the prover sending a large message to the verifiers, followed by the verifiers exchanging very small messages, we believe that the number of needed communication bits is a good proxy for future performance.

The existing descriptions of Sharemind's protocols make straightforward the computation of their execution and verification costs in terms of communicated bits. We have performed the computation for the protocols working with integers, and counted the number of bits that need to be delivered for executing and verifying an instance of the protocol. We have not taken into account the signatures, the broadcast overhead, and the final alleged zero hashes that the verifiers exchange, because these can be amortized over a large number of protocols executing either in parallel or sequentially.

Table 4.13 presents our findings. For each protocol, the results are presented in the form $\frac{x:y:z}{1:\boldsymbol{a}:b}$. The upper line lists the total communication cost (in bits): x for the execution of the protocol, y for its verification in the post-execution phase, and z for the generation of precomputed tuples in the preprocessing phase. The suffixes k and M denote the multipliers 10^3 and 10^6, respectively. The lower line is computed directly from the upper line, and it shows how many times more expensive each phase is, compared to the execution phase (i.e. $\boldsymbol{a} = y/x, b = z/x$). The most interesting value is $\boldsymbol{a}$ that shows how much overhead our verification gives in the online phase, compared to passively secure computation.

In estimating the costs of generating precomputed tuples, we have assumed the tuples to be generated in batches of 2^{20}, with security parameter $\eta = 80$. Section 4.3.2 describes the number of extra tuples that we must send for correctness checks. We consider the selected parameters rather conservative; we would need less extra tuples and less communication during the preprocessing phase if we increased the batch size or somewhat lowered the security parameter. Increasing the batch size to ca. 100 million would drop the parameter m from 5 to 4, thereby reducing the communication needs of preprocessing by 20%. If we take $\eta = 40$, then $m = 3$ would be sufficient.

The described integer protocols in Table 4.13 take inputs additively shared between three computing parties and deliver similarly shared outputs. In the "standard" protocol set, the available protocols include multiplication, division (with private or with public divisor), bit shifts (with private or public shift), comparisons and bit decomposition, for certain bit widths. We left out the protocols for operations that require no communication between parties during execution or verification phase: addition, and multiplication with a constant.

We see that the verification overhead (normalized to communication during the execution phase) of different protocols varies quite significantly. While most

Table 4.13: Communication overheads of integer operation verification

Operation	bit width		
	16	32	64
multiplication	96 : 384 : 2017 *1* : ***4*** : *21*	192 : 768 : 4034 *1* : ***4*** : *21*	384 : 1536 : 8067 *1* : ***4*** : *21*
division	9752 : 106.5k : 5.0M *1* : ***10*** : *514*	31.2k : 339.6k : 28.5M *1* : ***10*** : *914*	87.6k : 941.4k : 181.2M *1* : ***10*** : *2069*
div. with pub.	948 : 11.3k : 339.9k *1* : ***11*** : *359*	2180 : 26.1k : 1.3M *1* : ***11*** : *581*	4932 : 59.1k : 4.8M *1* : ***11*** : *982*
priv. $\ll$ priv.	400 : 5504 : 141.3k *1* : ***13*** : *353*	1296 : 21.2k : 1.1M *1* : ***16*** : *811*	4624 : 83.5k : 8.1M *1* : ***18*** : *1758*
priv. $\gg$ priv.	864 : 16.9k : 185.9k *1* : ***19*** : *215*	2352 : 52.9k : 314.0k *1* : ***22*** : *134*	7120 : 198.8k : 1.1M *1* : ***27*** : *161*
priv. $\gg$ pub.	468 : 4090 : 52.9k *1* : ***8*** : *113*	1092 : 9690 : 182.8k *1* : ***8*** : *167*	2564 : 22.4k : 658.2k *1* : ***8*** : *257*
equality	106 : 424 : 4549 *1* : ***4*** : *43*	218 : 872 : 14.3k *1* : ***4*** : *66*	442 : 1768 : 49.3k *1* : ***4*** : *112*
less than	719 : 7440 : 46.0k *1* : ***10*** : *64*	1750 : 18.7k : 127.3k *1* : ***10*** : *73*	4109 : 44.7k : 354.7k *1* : ***10*** : *86*
additive to xor	416 : 3008 : 18.1k *1* : ***7*** : *44*	1024 : 7552 : 49.4k *1* : ***7*** : *48*	2432 : 18.2k : 135.5k *1* : ***7*** : *56*
xor to additive	288 : 2144 : 14.7k *1* : ***7*** : *51*	1088 : 8384 : 58.7k *1* : ***7*** : *54*	4224 : 33.2k : 234.2k *1* : ***7*** : *55*

of the protocols require 7–20 times more communication during the verification phase than in the execution phase, the important case of integer multiplication has the overhead of only four times. Even more varied are the overheads for preprocessing, with integer multiplication again having the smallest overhead of 21 and the protocols working on smaller data having generally smaller overheads.

It is important to note that our goal was to optimize time and communication of the *verification* phase. If we wanted to optimize the total communication, including the preprocessing, we would possibly use some alternative approach. For example, instead of using trusted bits for bit decomposition, we could use AND triples, so that the verifiers could compute the decompositions themselves. Such optimizations are out of the scope of this thesis.

4.8 Summary

We have proposed a scheme transforming passively secure protocols with honest majority to covertly secure ones. The protocol transformation is suitable to be implemented on top of some existing, highly efficient, passively secure SMC frameworks, especially those that use 3 parties and computation over rings of size 2^m. The framework will retain its efficiency, as the time from starting a

computation to obtaining the result at the end of the execution phase will increase only slightly. We evaluated the verification on top of the Sharemind SMC framework and found its overhead to be of acceptable size, roughly an order of magnitude larger than the complexity of the SMC protocols themselves included in the framework, which are already practicable.

In general, we believe that in most situations, where sufficiently strong legal or contractual frameworks are in place, providing protection against covert adversaries is sufficient to cover possible active corruptions. The computing parties should have a contract describing their duties in place anyway [36], this contract can also specify appropriate punishments for being caught deviating from the protocol. By randomly deciding (with probability p) after a protocol run whether it should be verified, our method still achieves covert security, but the *average* overhead of verification is reduced by $1/p$ times. It is likely that overheads smaller than the execution time of the original passively secure protocol may be achieved in this manner, while keeping the consequences of misbehaving sufficiently severe. Auditability helps in setting up the contractual environment that establishes the consequences.

CHAPTER 5

PROTECTING DATA FROM HONEST PARTIES

5.1 Chapter Overview

Data is a toxic asset [96]. If it has been collected, then it has to be protected from breaches. Hence one should not collect data that is unnecessary or has little use. To make sure that one is not collecting such data, one should try to never learn that data in the first place.

In existing models of multiparty protocols, the security goals of a party are not violated if it learns too much: an honest party may simply ignore the messages not meant for it, or the data it has learned because of the misbehaviour of some other party. In practice, data erasure may be a complex and expensive process, involving thorough scrubbing or destruction of storage media.

An honest party's attempt to not learn the data that it is not supposed to learn brings about an adversarial goal that has not been considered so far. The adversary may deliberately try to cause an honest party to learn some other honest party's private data, making the second honest party's data derivable from the first honest party's view. The adversary's inability to learn such data itself does not imply the impossibility of such attacks. If some secret leaks from one honest user to another honest user, this secret may just remain unnoticed by the adversary.

As a practical illustration of this problem, let us take one real-world SMC project [12]. In the setup of this project, there have been three computing parties, two of whom have been separate governmental institutions (who definitely follow the protocol and do not collaborate), and the third one a private company that could be less trusted. In these settings, even if the private company was completely distrusted, in theory it would be sufficient to use protocols that tolerate one malicious party. However, in practice, the input parties would most probably not agree to run a protocol where misbehaviour of the private company leaks some

of their private data to the governmental institutions, even if the private company does not gain any information itself.

We give some examples of particular protocols suffering from such problems. In Chapter 4 we have proposed some protocols and proven that they are secure in the UC model. Regardless of being provably secure, there are still some problems with them, that may possibly prevent users from participating in such protocols.

- Problems of $\Pi_{transmit}$ (Figure 4.4): in this protocol, if communication between sender and receiver fails, then the sender is required to deliver the message to all parties, so that at least one of them would forward it to the receiver. However, honest parties are not supposed to learn that message in the ideal functionality $\mathcal{F}_{transmit}$ (Figure 4.3).

- Problems of Π_{commit} (Figure 4.17): if weak opening of a commitment x fails, strong opening requires to reveal up to $t-1$ shares of some other private values that have been used for computing x. These values are not supposed to be opened in the ideal functionality $\mathcal{F}_{commit}$ (Figure 4.16) and indeed, the corrupted parties get no more than $t-1$ shares. However, any honest party that is not involved in the conflict possesses one additional share, and so it may get t shares that are sufficient to reconstruct the secret.

These problems are not captured by the UC framework, since it assumes that there is a single monolithic adversary that controls all the corrupted parties. Construction of a simulator relies on the assumption that a value may indeed be leaked since the adversary knows it anyway. In practice, it may be still unpleasant to leak a secret value to some honest party even if some other corrupted party has already seen it. For example, if an attacker has broken into a user's mailbox, it still does not imply that the user is now ready to publish his e-mails to everyone since some attacker has seen them anyway.

If we care about the views of honest parties, we could treat each honest party as an independent adversary. There do exist some alternative definitions of UC that support multiple adversaries, such as CP (Collusion Preserving) computation or LUC (Local UC) described in Section 3.2. However, these models are too strong, and they are used to prove stronger properties that are not necessary for our purposes. If we treat each honest party as an adversary, we get a setting in which all parties are corrupted, and we immediately lose the advantage of assumptions on the number of corrupted parties, e.g. the honest majority assumption.

In this chapter, we study this problem more generally. We propose a model that is at least as strong as UC, and that additionally allows to detect if data are leaked to honest parties. We propose modifications to the protocols of Chapter 4, making them secure in this new model.

5.2 Attacks that We Want to Cover

The simplest way for the adversary to leak confidential data to an honest party is to send it through some side channels that are not related to the protocol. If we take such data into account, then we will be unable to securely implement e.g. secure (n, t)-threshold sharing assuming at most $t - 1$ corrupted parties. In particular, any coalition of $t - 1$ parties can always leak the secret to an honest party by sending to it the $t - 1$ shares that this coalition already has. If we reduce the number of corrupted parties by 1, then $t - 2$ parties will no longer be able to leak the secret to anyone. However, reducing the number of corrupted parties may lead to unrealistic assumptions. For example, in the 3-party case, $(3, 2)$-threshold sharing becomes possible only assuming that all parties are honest.

One possibility would be to give up and not use $(3, 2)$-threshold sharing in 3-party protocols. The problem is that, even if $(3, 2)$-threshold sharing is not used directly, many efficient 3-party protocols that assume one corrupted party [3, 16] require that the secret is being (temporarily) shared among two parties. If one of these parties is corrupted, it will immediately be able to leak its share to the other party using side channels. Even a *passively* corrupted party is able to do it, since protocol rules are not violated if side channels are used.

We think that, if confidential data leak to an honest party according to the protocol rules, this leakage is more significant than if the same data was sent by an untrusted corrupted party via side channels or even subliminal channels of the same protocol (e.g. sending it in place of some other messages or encoding the bits by message delays). First of all, an honest party is not supposed to be listening to any side channels. The adversary can still try to deliver information through subliminal channels of the same protocol that honest parties are obliged to execute in any case. However, the honest party may have no idea about the way it should interpret the information sent through subliminal channels, even if the secret is indeed encoded there. For example, if the adversary $\mathcal{A}_{active}$ succeeds in delivering $t-1$ shares to an honest party using a subliminal channel, that party will not even know that $\mathcal{A}_{active}$ is conducting such an attack. If later some other adversary $\mathcal{A}_{passive}$ gets access to that honest party's data, he will not know either how these shares should be recovered from the bit strings received by the honest party, and theoretically these bits can be reconstructed to an arbitrary value without knowing the particular strategy of $\mathcal{A}_{active}$.

In the problem of Π_{commit} mentioned in Section 5.1, the adversary $\mathcal{A}_{active}$ may cause all $t - 1$ shares of corrupted parties to be officially opened to all other parties. In that case, the honest party (and the adversary $\mathcal{A}_{passive}$) will definitely *know* that these particular $t-1$ opened values should be recombined with the share that the honest party already owns. Hence we treat exploiting such vulnerabilities as an attack.

As a summary, our model allows to detect threats where the computer of an honest user gets captured by an adversary $\mathcal{A}_{passive}$ who is completely independent from the adversary $\mathcal{A}_{active}$ actively attacking the protocol, and who only observes the honest user's data without trying to interfere with the protocol execution. This may represent the situation where the honest party itself is too curios, but it does not collaborate with the adversary $\mathcal{A}_{active}$ and tries to derive private information purely from its own data. It may also be the case that the honest party failed to clean its hard drive after running the protocol, and some intruder $\mathcal{A}_{passive}$, who has no relation with $\mathcal{A}_{active}$, has got access to its computer.

5.3 Weak Collusion Preservation

In this section we present a model that allows to formalize the attacks we described in Section 5.2. Two possible models from which we could start are LUC and CP (see Section 3.2). At first glance, the LUC approach seems more interesting since it clearly distinguishes the cases where a honest party P_i has received a message from another honest party P_j, or from a corrupted party P_k. However, many interesting properties are lost after splitting the adversaries to distinct coalitions. Hence we base our work on the collusion preserving (CP) computation. We call our model WCP (Weak Collusion Preservation).

5.3.1 Intuition

Before formally defining the WCP model, we describe the intuition behind it. We are looking for a model that would satisfy the following three properties.

Composability. We want to achieve composability like in the UC model. We state and prove this property in Theorem 5.1.

Implying UC. We do not want to lose any security properties that are already covered by the UC model. We want WCP-emulation to imply UC-emulation. We state and prove this property in Theorem 5.2.

Capturing information leakage to an honest party. The WCP model should be able to detect whether a protocol π leaks more information to honest parties than the ideal functionality $\mathcal{F}$ does. For simplicity, we define this property in the stand-alone model. The definition is based on indistinguishability between two games, depicted in Figure 5.1.

In the first game, the adversary attacks a real protocol π. The adversary consists of two isolated parts: $\mathcal{A}_{active}$ that interacts with the protocol in both directions (representing an active coalition) and $\mathcal{A}_{passive}$ that may only receive messages from π (representing an honest party). $\mathcal{A}_{active}$ chooses two inputs m_0 and m_1 for π, and sends m_0, m_1 to both the passive adversary $\mathcal{A}_{passive}$ and the challenger.

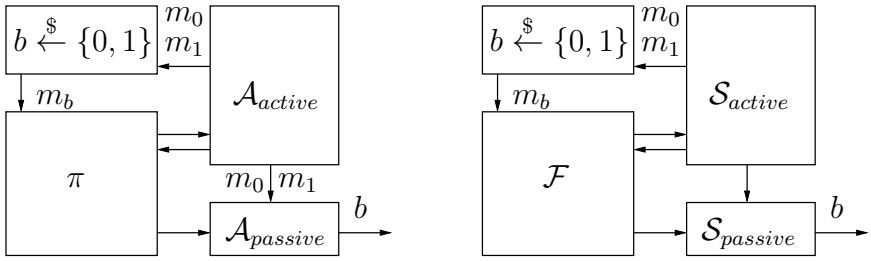


Figure 5.1: The games of leaking information to an honest party

The challenger generates $b \xleftarrow{\$} \{0,1\}$ and chooses the input m_b for π. The goal of $\mathcal{A}_{passive}$ is to guess b.

In the second game, simulator attacks an ideal functionality $\mathcal{F}$. Similarly to the first game, the simulator is split into two parts, $\mathcal{S}_{active}$ representing the active coalition, and $\mathcal{S}_{passive}$ representing an honest party. The guess is output by $\mathcal{S}_{passive}$. Since we do not treat existence of subliminal channels as a vulnerability, we allow $\mathcal{S}_{active}$ send to $\mathcal{S}_{passive}$ arbitrary messages, not only m_0 and m_1. Otherwise, $\mathcal{S}_{passive}$ would not be able to simulate e.g. some corrupted party leaving the protocol, which can be treated as one-bit subliminal channel.

In the stand-alone games, we do not need to constrain the attacks of $\mathcal{A}_{active}$ in any way. However, we might have problems when extending this definition to a composable one. The reason is that interesting UC-like composability properties rely on the fact that the *dummy* adversary (that does not do anything except forwarding the messages between the protocol and the environment $\mathcal{Z}$) is the strongest type of adversaries, delegating the attack to $\mathcal{Z}$. This property would not hold if $\mathcal{A}_{active}$ was able to use any information that $\mathcal{Z}$ does not possess, but at the same time we do not want $\mathcal{Z}$ to mix the views of $\mathcal{A}_{active}$ and $\mathcal{A}_{passive}$. For this reason, we put constraints on $\mathcal{A}_{active}$. It is still allowed to use the inputs of corrupted parties as well as any randomness generated by them. It may also use any messages that it receives from the party corrupted by $\mathcal{A}_{passive}$. However, in general, it will not use the messages that it receives from the *other honest parties* that are not corrupted neither by $\mathcal{A}_{active}$ nor $\mathcal{A}_{passive}$. This constraint may omit some attacks that correspond to our intuition. Nevertheless, the attacks described in Section 5.1 will still be captured.

The previous discussion is summarized in Definition 5.1. In Proposition 5.1, we state and prove that this definition is satisfied by our model.

Definition 5.1. Let π be a multiparty protocol, and $\mathcal{F}$ an ideal functionality. Consider the following two security games.

- $\mathsf{G}^{\mathcal{A}}_{real}$: The adversary $\mathcal{A}_{active}$ chooses two inputs m_0 and m_1 for π, and sends

m_0, m_1 to $\mathcal{A}_{passive}$ and the challenger. The challenger generates $b \xleftarrow{\$} \{0,1\}$ and chooses the input m_b for π. $\mathcal{A}_{passive}$ passively corrupts a single party P_k and receives all information known to P_k. $\mathcal{A}_{active}$ corrupts a subset of parties $\mathcal{C}$ in π, and it may choose messages for them. It receives from π all messages exchanged by the parties of $\mathcal{C}$, and the messages sent to $\mathcal{C}$ by P_k. After the protocol execution has been finished, $\mathcal{A}_{passive}$ makes a guess b' of b. $\mathsf{G}^{\mathcal{A}}_{real}$ outputs $b' = b$.

- $\mathsf{G}^{\mathcal{S}}_{ideal}$: The simulator $\mathcal{S}_{active}$ chooses two inputs m_0 and m_1 for $\mathcal{F}$, and sends m_0, m_1 to $\mathcal{S}_{passive}$ and the challenger. The challenger generates $b \xleftarrow{\$} \{0,1\}$ and chooses the input m_b for $\mathcal{F}$. $\mathcal{S}_{active}$ corrupts a subset of parties in $\mathcal{F}$. It may receive messages from $\mathcal{F}$ and choose messages for the corrupted parties in $\mathcal{F}$. The simulator $\mathcal{S}_{passive}$ passively corrupts a single party and receives all messages known to that party. After the protocol execution has been finished, $\mathcal{S}_{passive}$ makes a guess b' of b. $\mathsf{G}^{\mathcal{S}}_{ideal}$ outputs $b' = b$.

We say that the protocol π leaks to honest parties as much information as $\mathcal{F}$ if $\left|\mathbf{Pr}\left[\mathsf{G}^{\mathcal{A}}_{real} = 1\right] - \mathbf{Pr}\left[\mathsf{G}^{\mathcal{S}}_{ideal} = 1\right]\right| \leq \varepsilon$ for a negligible ε.

5.3.2 Definitions

We adjust the definitions of UC and CP to the new model WCP. We base our work on the collusion preserving (CP) computation of [1] described in Section 3.2.1. In the CP model, all the adversaries are connected with the environment $\mathcal{Z}$. Hence if we use CP in a straightforward way, then $\mathcal{Z}$ gets the values of corrupted parties as well as the values of all the honest parties, and that is not what we would expect from the attacks described in Section 5.2. We need to modify the construction in such a way that it would take into account that the honest parties will never use $\mathcal{Z}$ to share their view with the corrupted parties. Instead of assigning an adversary to each *party*, we assign an adversary to each *coalition*. We put some additional constraints on the adversary that ensure that the outputs of *only one* of these coalitions reach the environment.

Definition 5.2 (t-coalition split adversary)**.** Let n be the number of parties. A t-coalition split adversary $\mathcal{A}$ is a set of PPT machines $\{\mathcal{A}^H_1, \ldots, \mathcal{A}^H_n, \mathcal{A}^H, \mathcal{A}^L\}$ defined as follows.

1. The connections between $\mathcal{A}$, $\mathcal{Z}$, and the protocol ϕ, as well as the connections between different components of $\mathcal{A}$, are depicted on the left hand side of Figure 5.2. Any communication inside $\mathcal{A}$ goes either from $\mathcal{A}^L$ to $\mathcal{A}^H_i$, from $\mathcal{A}^H_i$ to $\mathcal{A}^H$, or from $\mathcal{A}^H$ to $\mathcal{A}^L$. The machine $\mathcal{A}^H$ mediates the

communication between $\mathcal{Z}$ and all the other machines $\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^L$. For all $i \in [n]$, the machine $\mathcal{A}_i^H$ [resp. $\mathcal{A}^L$] do not receive inputs from $\mathcal{A}^H$ [resp. π] nor give outputs to π [resp. $\mathcal{A}^H$].

2. All party corruptions are arranged by $\mathcal{A}^L$. Its construction should ensure that each party is corrupted by exactly one adversary $\mathcal{A}_i^H$. The adversary $\mathcal{A}_1^H$ actively corrupts up to t parties, and each other adversary passively corrupts at most one party. $\mathcal{A}^L$ may send messages to all parties.

3. There is some $j \in [n]$, such that for all $i \in [n] \backslash \{j\}$, the internal state and the behaviour of $\mathcal{A}^H$ do not depend on the inputs coming from $\mathcal{A}_i^H$. We call $\mathcal{A}_j^H$ the *true* adversary and the other $\mathcal{A}_i^H$-s the *false* adversaries.

Let $k = \mathsf{true}(\mathcal{A})$ be the adversary index k such that $\mathcal{A}_k^H$ is the true adversary.

The property (1) lets the information moving from $\mathcal{Z}$ to π to be controlled by a single adversary $\mathcal{A}^L$, and it splits the information moving from π to $\mathcal{Z}$ among different receiving adversaries. The property (2) constructs an actively corrupted coalition of size at most t, and lets each honest party be controlled by a separate passive adversary. The property (3) guarantees that the views of different coalitions will not be merged.

Referring to the intuitive description of Section 5.3.1, $\mathcal{A}^L$ corresponds to the adversary $\mathcal{A}_{active}$ that attacks the protocol. The adversary $\mathcal{A}_1^H$ corresponds to the same actively corrupted coalition; this is not covered by the game of Figure 5.1, and it is needed to make WCP as strong as UC, taking into account the information that the active attacker has learned itself. Each other adversary $\mathcal{A}_i^H$ for $i \neq 1$ passively corrupts a single party and corresponds to $\mathcal{A}_{passive}$ of Section 5.3.1.

We could define WCP emulation analogously to CP by replacing *any* adversary with a *t-coalition split* adversary. However, we now need to be careful with the simulator definition. If we allow $\mathcal{S}$ to be an arbitrary PPT machine, then it may happen that $(S \| \mathcal{A})$ is no longer a t-coalition split adversary. Hence we need to constrain the class of simulators.

Definition 5.3 (split simulator)**.** Let $\mathcal{A} = \{\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^H, \mathcal{A}^L\}$ be a t-coalition split adversary. A split simulator $\mathcal{S}_k = \{\mathcal{S}_1^H, \ldots, \mathcal{S}_n^H, \mathcal{S}^H(k), \mathcal{S}^L\}$ is a set of PPT ITMs defined as follows.

1. The connections between $\mathcal{S}_k$, $\mathcal{A}$, and the protocol ϕ, as well as the connections between different components of $\mathcal{A}$, are depicted on the right hand side of Figure 5.2. The communication inside $\mathcal{S}_k$ goes either from $\mathcal{S}^L$ to $\mathcal{S}_i^H$, from $\mathcal{S}_i^H$ to $\mathcal{S}^H(k)$, or from $\mathcal{S}^H(k)$ to $\mathcal{S}^L$. The machine $\mathcal{S}^H(k)$ mediates the communication between $\mathcal{A}$ and all the other machines $\mathcal{S}_1^H, \ldots, \mathcal{S}_n^H, \mathcal{S}^L$. For all $i \in [n]$, the machine $\mathcal{S}_i^H$ [resp. $\mathcal{S}^L$] does not receive inputs from $\mathcal{S}^H(k)$ [resp. ϕ] nor gives outputs to ϕ [resp. $\mathcal{S}^H(k)$].

2. $\mathcal{S}^H(k)$ is an ITM with the following fixed behaviour:

   - For all messages m coming from $\mathcal{A}^L$, it sends (L, m) to $\mathcal{S}^L$.
   - On input m from $\mathcal{S}_i^H$, it forwards m to $\mathcal{A}_i^H$.
   - On input (L, m) from $\mathcal{S}_k^H$, it sends (k, m) to $\mathcal{S}^L$.

   In this way, we let $\mathcal{S}^H(k)$ depend on the knowledge which adversary $\mathcal{A}_k^H$ is the true one. Here L is just a fixed symbol used to distinguish the messages.

The property (1) is analogous to the similar property of t-coalition split adversary, letting the information moving from $\mathcal{A}$ to ϕ to be controlled by a single simulator $\mathcal{S}^L$, and splitting the information moving from ϕ to $\mathcal{A}$ among different receiving simulators $\mathcal{S}_i^H$, so that each simulator has the view of its own coalition. The property (2) guarantees that the simulators $\mathcal{S}_i^H$ do not share any information with each other, and $\mathcal{S}_i^H$ does the simulation only for $\mathcal{A}_i^H$.

We show that $(\mathcal{S}_{\mathsf{true}(\mathcal{A})} \| \mathcal{A})$ is also a t-coalition split adversary. Otherwise it may happen that we give more power to the adversary that attacks an ideal functionality than to the adversary that attacks a real functionality, and that would result in weaker security proofs.

Lemma 5.1. *Let $\mathcal{A} = \{\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^H, \mathcal{A}^L\}$ be a t-coalition-split adversary, and let $\mathcal{S}_k = \{\mathcal{S}_1^H, \ldots, \mathcal{S}_n^H, \mathcal{S}^H(k), \mathcal{S}^L\}$ be a split simulator. Then $(\mathcal{S}_{\mathsf{true}(\mathcal{A})} \| \mathcal{A})$ is also a t-coalition split adversary.*

Proof. By Definition 5.2, there exist channels only from $\mathcal{A}^L$ to $\mathcal{A}_i^H$ for all $i \in [n]$, but not the other way around. By Definition 5.3, using $\mathcal{S}^H$ as a mediator, $\mathcal{S}_i^H$ delivers messages from the protocol π to $\mathcal{A}_i^H$, and $\mathcal{S}^L$ delivers messages from $\mathcal{A}^L$ to π. Similarly, there exist channels only from $\mathcal{S}^L$ to $\mathcal{S}_i^H$ for all $i \in [n]$, and only the messages of $\mathcal{S}_k^H$ s.t $\mathcal{A}_k^H$ is the true adversary may reach $\mathcal{S}^L$. We may define $\mathcal{A}_i'^H = (\mathcal{S}_i^H \| \mathcal{A}_i^H)$, $\mathcal{A}'^H = (\mathcal{S}^H \| \mathcal{A}^H)$, and $\mathcal{A}'^L = (\mathcal{S}^L \| \mathcal{A}^L)$, where $\mathcal{S}_i^H$ and $\mathcal{A}_i^H$ [resp. $\mathcal{S}^L$ and $\mathcal{A}^L$] communicate directly instead of using $\mathcal{S}^H$ as a mediator. In order to forward messages from $\mathcal{S}_k^H$ to $\mathcal{S}^L$, we allow $\mathcal{S}_i^H$ to send messages directly to the component $\mathcal{S}^H$ of $\mathcal{A}'^H$ that forwards the messages of $\mathcal{S}_k^H$ them to the component $\mathcal{S}^L$ of $\mathcal{A}'^L$.

As the result, direct communication takes place only from $\mathcal{A}'^L$ to $\mathcal{A}_i'^H$, but not the other way around. $\mathcal{A}'^L$ does not receive inputs from π, and none of the $\mathcal{A}_i'^H$ outputs to π. Only the messages of the true adversary $\mathcal{A}_k'^H$ are forwarded by $\mathcal{A}'^H$ to $\mathcal{A}'^L$. The resulting adversary $\mathcal{A}_S{}' = \{\mathcal{A}_1'^H, \ldots, \mathcal{A}_n'^H, \mathcal{A}'^H, \mathcal{A}'^L\}$ satisfies the definition of a t-coalition split adversary. □

We may now define WCP emulation similarly to UC emulation (Definition 2.1). Since the number of actively corrupted parties, t, is a part of the adversary definition, we need to parametrize the definition of emulation with t. We call it a t-WCP emulation.

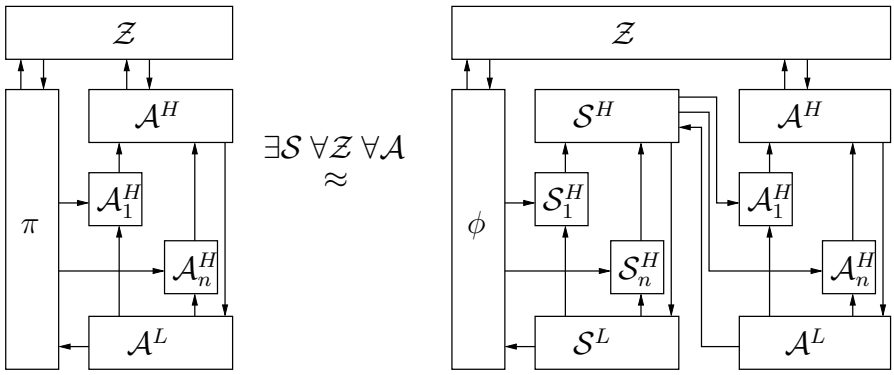


Figure 5.2: The protocol π t-WCP emulates the protocol ϕ

Definition 5.4 (t-WCP emulation)**.** Let π and ϕ be n-party protocols. We say that the protocol π t-WCP-emulates the protocol ϕ if there exists a PPT split simulator $\mathcal{S}_k = \{\mathcal{S}_1^H, \ldots, \mathcal{S}_n^H, \mathcal{S}^H(k), \mathcal{S}^L\}$, such that, for any PPT t-coalition split adversary $\mathcal{A} = \{\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^H, \mathcal{A}^L\}$, and for any PPT environment $\mathcal{Z}$, for a t-coalition split adversary $\mathcal{A}_S = (\mathcal{S}_{\mathsf{true}(\mathcal{A})} \| \mathcal{A})$, the probability ensembles $EXEC_{\pi,\mathcal{A},\mathcal{Z}}$ and $EXEC_{\phi,\mathcal{A}_S,\mathcal{Z}}$ are indistinguishable.

The definition is correct by Lemma 5.1. It is depicted in Figure 5.2. The definitions of a protocol π WCP-realizing an ideal functionality $\mathcal{F}$, as well as protocol emulation in hybrid model, can be derived from this definition similarly to UC model.

We could as well define blackbox simulatability, constructing the adversary $\mathcal{A}_S = \{\mathcal{S}_1^H(\mathcal{A}_1^H), \ldots, \mathcal{S}_n^H(\mathcal{A}_n^H), \mathcal{S}^H(\mathcal{A}^H), \mathcal{S}^L(\mathcal{A}^L)\}$. In this case, $\mathcal{S}^H$ gets the parameter $k = \mathsf{true}(\mathcal{A})$ directly from the description of $\mathcal{A}^H$.

5.3.3 Technical Details

Communication between corrupted parties and the adversary. In UC, the easiest way to model the corrupted parties is to let all their messages be chosen by $\mathcal{A}$. When a corrupted party P_i is supposed to send a message m to another party P_j, then $\mathcal{A}$ chooses m^* that should be sent to P_j instead. It is possible that $m \neq m^*$. We could do the same in WCP model, letting $\mathcal{A}^L$ to choose the message for P_i. If $\mathcal{A}_1^H$ is the true adversary, then $\mathcal{A}$ is as strong as a monolithic adversary that corrupts the same parties as $\mathcal{A}_1^H$ does (as we prove later in Section 5.3.7). However, it may weaken the adversary in the cases where we treat some *passive* $\mathcal{A}_k^H$ as the real one. In this case, the messages received by the active adversary $\mathcal{A}_1^H$ will not be accepted by $\mathcal{A}^H$.

The first problem is that $\mathcal{A}^L$ might not know that P_i is waiting for an input,

since only $\mathcal{A}_1^H$ has access to the current state of P_i. To solve it, we may assume that P_i processes the messages of $\mathcal{A}^L$ in the order of coming, and waits until the next message comes from $\mathcal{A}^L$. Also, $\mathcal{A}^L$ may synchronize itself with $\mathcal{A}_k^H$ and $\mathcal{Z}$.

The second problem is that $\mathcal{A}^L$ does not get the messages received by actively corrupted parties, and hence does not know how they should respond if they want to follow the protocol. To solve this problem, we allow $\mathcal{A}^L$ to send to an actively corrupted P_i a dummy message $\top$, denoting that P_i should compute m according to the protocol rules. To ensure that all simulations would succeed, we will need to define ideal functionalities in such a way that they also admit $\top$. If $\mathcal{A}^L$ wants P_i to remain silent, let the dummy message $\bot$ be chosen for P_i.

The behaviour of corrupted P_i can be summarized as follows:

- The corruption of P_i by the coalition $\mathcal{A}_j^H$ is determined by $\mathcal{A}^L$, who sends a message $(\mathsf{corrupt}, j)$ to the party P_i. After the machine P_i receives that message, it forwards its internal state and all further received messages to the adversary $\mathcal{A}_j^H$.

- At any time when P_i should send a message m to another party P_j, it reads the next message (j, m^*) from $\mathcal{A}^L$, and sends m^* to P_j instead. If $m^* = \bot$, then P_j does not send anything to P_j. If $m^* = \top$, then P_j sends to P_j the message m computed according to the protocol rules. Namely, it computes m correctly according to its inputs, randomness, and the messages received from the other parties so far (these received messages may be erroneous if P_j or some other party misbehaved on earlier steps of the protocol).

Similarly to Chapter 4, we will often use an informal expression "x is chosen by ${\mathcal{A}_S}^L$" in definitions of ideal functionalities, where a message of the form $(command, id, x)$ comes from a corrupted party. Formally, in such cases the ideal functionality $\mathcal{F}$ sends a message $(\mathsf{arrived}(command), id, x)$ to ${\mathcal{A}_S}_i^H$, a message $(\mathsf{arrived}(command), id)$ to all ${\mathcal{A}_S}_j^H$ for $j \neq i$ (to solve the synchronicity problems between $\mathcal{A}_i^H$ and $\mathcal{A}^L$), and waits until ${\mathcal{A}_S}^L$ sends $(\mathsf{change}(command), id, x')$, so that $\mathcal{F}$ will further use x' instead of x. If $x' = \top$, then $\mathcal{F}$ just takes x.

Simplifications due to one true adversary. In an extreme case, there may be n distinct adversaries $\mathcal{A}_i^H$, one for each of the n parties. This requires n different simulators $\mathcal{S}_i^H$, and reasoning about their joint view makes the proofs rather complicated. Since there is always exactly one true adversary $\mathcal{A}_k^H$, and the outputs of all other $\mathcal{A}_i^H$ are ignored by $\mathcal{A}^H$, regardless of the values they get from $\mathcal{S}_i^H$, it suffices to observe the joint views of the pairs $(\mathcal{S}^L, \mathcal{S}_k^H)$ for $k \in [n]$ separately. The behaviour of the machine $\mathcal{S}^H(k)$ is fixed, so it does not need to be defined. Moreover, since we are only interested in simulating correct view of

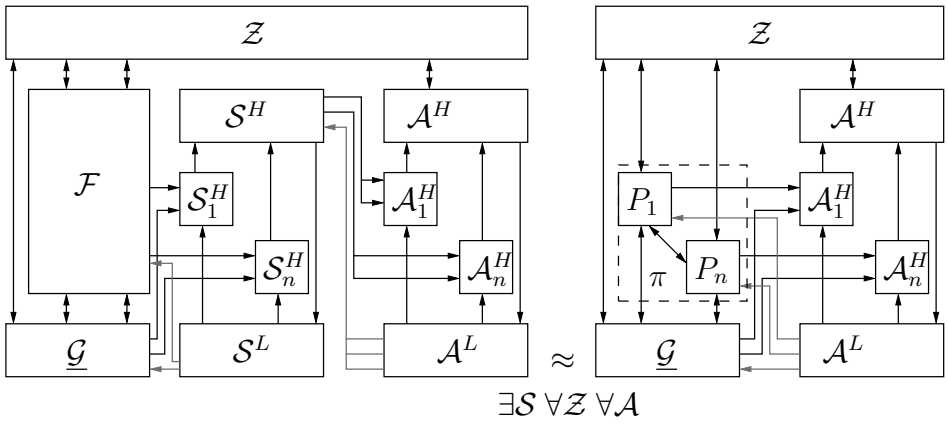


Figure 5.3: The protocol π $\underline{\mathcal{G}}$-WCP realizes the ideal functionality $\mathcal{F}$

the true adversary, we may assume in the proofs that $\mathcal{S}^L$ gets all messages of $\mathcal{S}_k^H$ through $\mathcal{S}^H(k)$. In this way, the entire simulation can be delegated to $\mathcal{S}_k^H$ alone, who is able to exchange messages between ϕ and $\mathcal{A}$ in both directions, and $\mathcal{S}^L$ and $\mathcal{S}^H$ do not need to be used explicitly in the proofs of WCP emulation. Depending on which adversary is the true one, each proof in t-WCP model consists of the following two types of proofs:

- $\mathcal{A}_1^H$ is the true adversary, and the active attacker tries to get information itself (this makes WCP it as strong as UC).
- $\mathcal{A}_k^H$ for $k \neq 1$ is the true adversary, and the attacker tries to leak information to the honest party corrupted by $\mathcal{A}_k^H$ (this allows to discover additional attacks not covered by UC).

If the protocol is symmetric w.r.t. all parties, then it is sufficient to describe the behaviour just for two types of $\mathcal{S}_k^H$, one for an active adversary, and one for a passive adversary.

5.3.4 Relations with Generalized Universal Composability

The CP model is actually based not on UC, but GUC (see Section 2.2.3), which allows a protocol π to use a shared functionality $\underline{\mathcal{G}}$ whose party ports may be accessed not only by π (like in UC), but also by $\mathcal{Z}$. We may extend this notion to WCP. The shared functionality $\underline{\mathcal{G}}$ interacts with $\mathcal{A}$ in exactly the same way as an ordinary functionality $\mathcal{F}$ does, receiving inputs from $\mathcal{A}^L$ and sending outputs to $\mathcal{A}_i^H$. The pictorial representation of $\underline{\mathcal{G}}$-GWCP realization is given in Figure 5.3, where for simplicity each adversary corrupts one party.

5.3.5 Capturing Information Leakage to an Honest Party

Compared to the Definition 5.1 that describes our intuition behind the security model, Definitions 5.2 and 5.4 are more complicated. We need to show that, if a protocol π WCP-realizes an ideal functionality $\mathcal{F}$, then the protocol π leaks to honest party as much information as $\mathcal{F}$ does according to Definition 5.1.

Proposition 5.1. Let π be a protocol that t-WCP-realizes an ideal functionality $\mathcal{F}$. Assuming that $\mathcal{A}_{active}$ corrupts at most t parties, the protocol π leaks to honest party as much information as $\mathcal{F}$ does according to Definition 5.1.

Proof. Suppose by contrary that there is an adversary pair $(\mathcal{A}_{active}, \mathcal{A}_{passive})$ such that difference of the success probabilities in the games $\mathsf{G}^{\mathcal{A}}_{real}$ and $\mathsf{G}^{\mathcal{S}}_{ideal}$ of Definition 5.1 is non-negligible for any choice of simulators $\mathcal{S}_{active}$ and $\mathcal{S}_{passive}$. We use it to construct a t-coalition split adversary $\mathcal{A} = \{\mathcal{A}^H_1, \ldots, \mathcal{A}^H_n, \mathcal{A}^H, \mathcal{A}^L\}$ that breaks WCP-emulation. First, we take $\mathcal{A}^H_k := \mathcal{A}_{passive}$ for an arbitrary $k \neq 1$, letting the final guess of $\mathcal{A}_{passive}$ be output to $\mathcal{Z}$, and define $\mathsf{true}(\mathcal{A}) := k$. We then take $\mathcal{A}^L := \mathcal{D}$ and $\mathcal{A}^H_i := \mathcal{D}$ for all $i \neq k$, where $\mathcal{D}$ is just a message forwarding box. We remove the implicit channel from $\mathcal{A}^H_k$ to $\mathcal{A}^L$ (the one arranged by $\mathcal{A}^H$), as there is no channel from $\mathcal{A}_{passive}$ to $\mathcal{A}_{active}$.

The environment $\mathcal{Z}$ absorbs $\mathcal{A}_{active}$. It chooses the same inputs m_0 and m_1 that guaranteed success in distinguishing the games $\mathsf{G}^{\mathcal{A}}_{real}$ and $\mathsf{G}^{\mathcal{S}}_{ideal}$, and chooses a random m_b as the input for the protocol (π or $\mathcal{F}$). In addition, it forwards to $\mathcal{A}^L$ all messages that it receives from $\mathcal{A}^H_k$, marking them with a special symbol to make them distinguishable from the other messages. The final output of $\mathcal{Z}$ is $b' = b$, where the guess b' is generated by $\mathcal{A}^H_k$. We get $\mathsf{G}^{\mathcal{A}}_{real} \approx EXEC_{\pi,\mathcal{A},\mathcal{Z}}$.

Assuming that π t-WCP-realizes $\mathcal{F}$, there is a simulator $\mathcal{S}_k = \{\mathcal{S}^H_1, \ldots, \mathcal{S}^H_n, \mathcal{S}^H(k), \mathcal{S}^L\}$ such that $|EXEC_{\mathcal{F},(\mathcal{S}_k\|\mathcal{A}),\mathcal{Z}} - EXEC_{\pi,\mathcal{A},\mathcal{Z}}| < \varepsilon$ for a negligible ε. We would like to take $\mathcal{S}_{active} := (\mathcal{S}^L \| \mathcal{A}^L)$ and $\mathcal{S}_{passive} := (\mathcal{S}^H_k \| \mathcal{A}^H_k, \mathcal{S}^H \| \mathcal{A}^H)$, getting the same interaction with $\mathcal{F}$ as in the WCP model. However, the problem is that, although we have removed the channel from $\mathcal{A}^H_k$ to $\mathcal{A}^L$, there may still be a channel from $\mathcal{S}^H_k$ to $\mathcal{S}^L$. Hence we adjust the definition of $\mathcal{A}$ and let $\mathcal{A}^H_k$ forward to $\mathcal{Z}$ all messages that it receives from $\mathcal{S}^H_k$. After getting these messages from $\mathcal{Z}$, $\mathcal{A}^L$ will forward them to $\mathcal{S}^L$. Again, special marking can be used to make these messages distinguishable from the other messages. In this way, $\mathcal{S}_k$ is a suitable simulator against this particular $\mathcal{A}$ even if we remove the communication channel from $\mathcal{S}^H_k$ to $\mathcal{S}^L$. Now $\mathcal{S}_{active} := (\mathcal{S}^L \| \mathcal{A}^L)$ and $\mathcal{S}_{passive} := (\mathcal{S}^H_k \| \mathcal{A}^H_k, \mathcal{S}^H \| \mathcal{A}^H)$ satisfy Definition 5.1, and we have $\mathsf{G}^{\mathcal{S}}_{ideal} \approx EXEC_{\mathcal{F},(\mathcal{S}_k\|\mathcal{A}),\mathcal{Z}}$.

Since we assumed by contrary that the difference of success probabilities in the games $\mathsf{G}^{\mathcal{A}}_{real}$ and $\mathsf{G}^{\mathcal{S}}_{ideal}$ is non-negligible, it should be $|EXEC_{\mathcal{F},(\mathcal{S}_k\|\mathcal{A}),\mathcal{Z}} - EXEC_{\pi,\mathcal{A},\mathcal{Z}}| > \varepsilon$, contradicting $|EXEC_{\mathcal{F},(\mathcal{S}_k\|\mathcal{A}),\mathcal{Z}} - EXEC_{\pi,\mathcal{A},\mathcal{Z}}| < \varepsilon$. □

5.3.6 Composition Theorem

Dummy Lemma

The proof of UC composition theorem is simpler if, instead of an arbitrary adversary $\mathcal{A}$, we consider the dummy adversary $\mathcal{D}$ that only forwards the messages between the protocol and the environment. This kind of adversary is in some sense the strongest one since it delegates all the attacks to the environment $\mathcal{Z}$, and it just gives to $\mathcal{Z}$ the entire view of the corrupted parties. This property of $\mathcal{D}$ is stated as the *dummy lemma*, that has been proven for UC in [21], and holds also in the LUC and CP models. In our WCP model we could also substitute the true adversary with a dummy adversary, similarly to UC. However, the false adversaries are not allowed to forward the messages. If we replace a false adversary with $\mathcal{D}$, it will be too strong since the view of the false adversary will be forwarded to the environment. We conclude that the dummy lemma of UC that works also for CP and LUC is not directly applicable to WCP. Nevertheless, it holds if $\mathcal{D}$ satisfies the t-coalition adversary definition.

Definition 5.5 (k-dummy t-coalition split adversary)**.** Let n be the number of parties, and let $k \in [n]$ be any fixed adversary index. The k-dummy t-coalition split adversary $\mathcal{D}\mathsf{k} = \{\mathcal{D}\mathsf{k}_1^H, \ldots, \mathcal{D}\mathsf{k}_n^H, \mathcal{D}\mathsf{k}^H, \mathcal{D}\mathsf{k}^L\}$ is a t-coalition split adversary, where:

- $\mathcal{D}\mathsf{k}^L$ is forwarding messages from $\mathcal{D}\mathsf{k}^H$ to the protocol;
- $\mathcal{D}\mathsf{k}_i^H$ for all $i \in [n]$ are forwarding messages from the protocol to $\mathcal{D}\mathsf{k}^H$;
- $\mathcal{D}\mathsf{k}^H$ is forwarding messages from $\mathcal{D}\mathsf{k}_k^H$ to $\mathcal{Z}$, and from $\mathcal{Z}$ to $\mathcal{D}\mathsf{k}^L$.

The definition is correct since $\mathcal{D}\mathsf{k}^L$ and $\mathcal{D}\mathsf{k}_i^H$ are clearly instances of $\mathcal{A}^L$ and $\mathcal{A}_i^H$ respectively, and although $\mathcal{D}\mathsf{k}^H$ does not forward messages from $\mathcal{D}\mathsf{k}_k^H$ to $\mathcal{D}\mathsf{k}^L$, it can be seen as an instance of a t-coalition split adversary where $\mathcal{A}^L$ ignores the inputs that it gets via $\mathcal{D}\mathsf{k}^H$ from $\mathcal{D}\mathsf{k}_k^H$. For n parties, there can be up to n different k-dummy adversaries $\mathcal{D}\mathsf{1}, \ldots, \mathcal{D}\mathsf{n}$, such that $\mathsf{true}(\mathcal{D}\mathsf{k}) = k$.

Lemma 5.2 (t-dummy lemma)**.** *Let π and ϕ be n-party protocols. Then π t-WCP-emulates ϕ according to Definition 5.4 if and only if it t-WCP-emulates ϕ with respect to all k-dummy t-coalition split adversaries for all $k \in [n]$.*

Proof. One proof direction is trivial since a t-dummy adversary is just an instance of a t-coalition split adversary. The other direction is more interesting. Let $\mathcal{S}_k$ be the split simulator for the k-dummy adversary $\mathcal{D}\mathsf{k}$ guaranteed by Definition 5.4 (that is, $\mathcal{S}_k$ satisfies $EXEC_{\phi,(\mathcal{S}_k \| \mathcal{D}\mathsf{k}),\mathcal{Z}'} \approx EXEC_{\pi,\mathcal{D}\mathsf{k},\mathcal{Z}'}$ for all $\mathcal{Z}'$.) We show that π t-WCP emulates ϕ according to Definition 5.4. We claim that $\mathcal{S}_{\mathsf{true}(\mathcal{A})}$ is a suitable simulator for an arbitrary t-coalition split adversary $\mathcal{A}$.

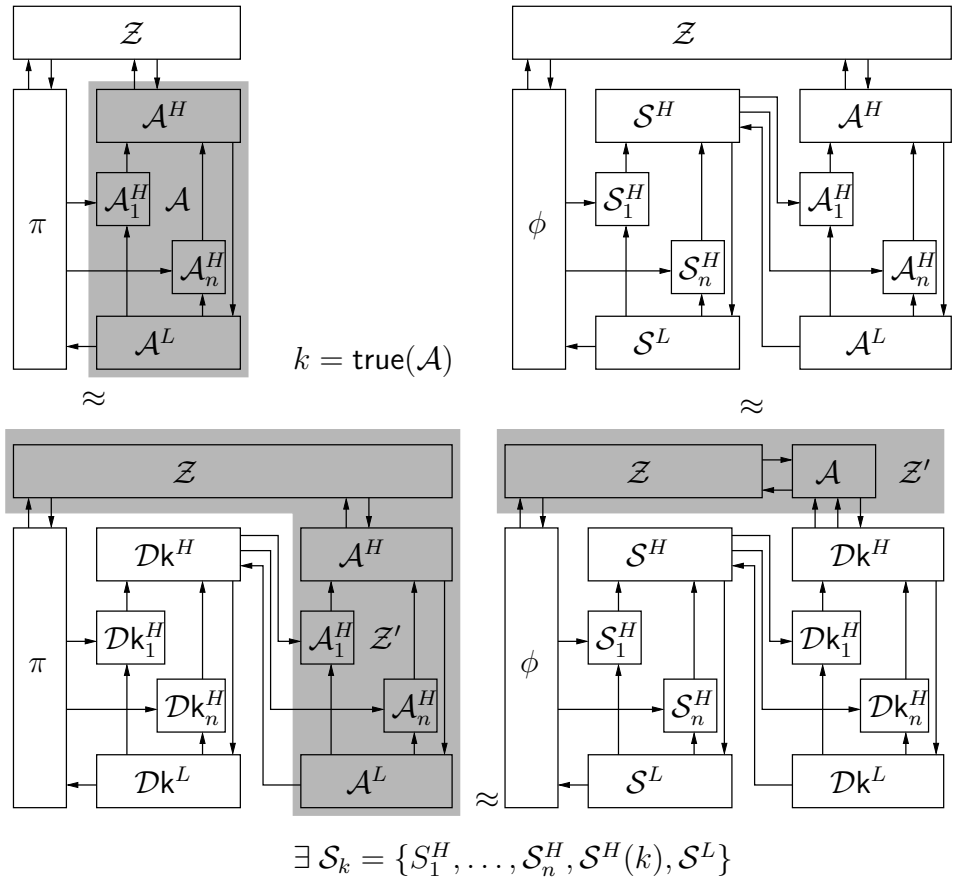


Figure 5.4: t-dummy lemma

Assume by contrary that there is a t-coalition split adversary $\mathcal{A}$ and an environment $\mathcal{Z}$ such that $\mathsf{true}(\mathcal{A}) = k$, and $|EXEC_{\phi,(\mathcal{S}_k\|\mathcal{A}),\mathcal{Z}} - EXEC_{\pi,\mathcal{A},\mathcal{Z}}| \geq \varepsilon$. We use it to construct $\mathcal{Z}'$ such that $|EXEC_{\phi,(\mathcal{S}_k\|\mathcal{D}\mathsf{k}),\mathcal{Z}'} - EXEC_{\pi,\mathcal{D}\mathsf{k},\mathcal{Z}'}| \geq \varepsilon$. First, we define $\mathcal{Z}' = (\mathcal{A}\|\mathcal{Z})$. Since the inner state of $\mathcal{A}^H$ does not depend on the inputs coming from $\mathcal{A}_i^H$ for $i \neq k$ anyway, we could as well put $\mathcal{D}\mathsf{k}$ between the protocol π [resp. the simulator $\mathcal{S}$] and $\mathcal{A}$, forwarding all messages from $\mathcal{A}^L$ to π [resp. $\mathcal{S}^L$], and delivering only the messages of parties corrupted by $\mathcal{A}_k^H$ [resp. of $\mathcal{S}_k^H$] to $\mathcal{A}_k^H$. We get $|EXEC_{\phi,(\mathcal{S}_k\|\mathcal{D}\mathsf{k}),\mathcal{Z}'} - EXEC_{\pi,\mathcal{D}\mathsf{k},\mathcal{Z}'}| \geq \varepsilon$. □

The quantities used in the proof are depicted in Figure 5.4. Note that $(\mathcal{S}_k\|\mathcal{D}\mathsf{k}) \approx \mathcal{S}_k$, since $\mathcal{D}\mathsf{k}$ just forwards messages from $\mathcal{S}_k^H$ to $\mathcal{Z}$ and from $\mathcal{Z}$ to $\mathcal{S}^L$, which could be as well handled by $\mathcal{S}^H$ alone.

WCP Composition Theorem

We want to show that protocols of the WCP model are composable. We state and prove a theorem that is very similar to UC composition (Theorem 2.1).

Theorem 5.1 (WCP composition theorem)**.** *Let ρ, ϕ, π be protocols such that ρ uses ϕ as subroutine, and π t-WCP-emulates ϕ. Then protocol $\rho[\phi \to \pi]$ t-WCP-emulates ρ.*

Proof. As the basis for our proof, we take the simpler proof variant of UC composition theorem of [21] that proves the claim for one instance of ϕ and then extends it to polynomially many calls of ϕ by induction (taking into account that simulation quality is lost). Similarly to the proofs of analogous theorems for UC, CP, LUC, we base our proof on t-dummy lemma.

Consider the protocol ρ that uses ϕ as subroutine. We need to prove that there exists a split simulator $\mathcal{S}_k = \{\mathcal{S}_1^H, \ldots, \mathcal{S}_n^H, \mathcal{S}^H(k), \mathcal{S}^L\}$ such that, for any t-coalition split adversary $\mathcal{A} = \{\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^H, \mathcal{A}^L\}$ and any environment $\mathcal{Z}$ we have $EXEC_{\rho[\phi\to\pi],\mathcal{A},\mathcal{Z}} \approx EXEC_{\rho,(\mathcal{S}_{\mathsf{true}(\mathcal{A})}\|\mathcal{A}),\mathcal{Z}}$. By Lemma 5.2, it suffices to prove that $EXEC_{\rho[\phi\to\pi],\mathcal{D}\mathsf{k},\mathcal{Z}} \approx EXEC_{\rho,(\mathcal{S}_k\|\mathcal{D}\mathsf{k}),\mathcal{Z}}$, i.e. $EXEC_{\rho[\phi\to\pi],\mathcal{D}\mathsf{k},\mathcal{Z}} \approx EXEC_{\rho,\mathcal{S}_k,\mathcal{Z}}$, for all $k \in [n]$.

Let $k \in [n]$ be arbitrary. By definition, the adversary $\mathcal{D}\mathsf{k}$ just forwards the messages between $\rho[\phi \to \pi]$ and $\mathcal{Z}$. There could be as well two instances of adversaries $\mathcal{D}\mathsf{k}$, one mediating the communication between $\mathcal{Z}$ and ρ (let it be denoted $\mathcal{D}\mathsf{k}^\rho$), and the other between $\mathcal{Z}$ and π (let it be denoted $\mathcal{D}\mathsf{k}^\pi$). We may write $\mathcal{D}\mathsf{k} = (\mathcal{D}\mathsf{k}^\phi\|\mathcal{D}\mathsf{k}^\rho)$ Taking $\mathcal{Z}' = (\mathcal{D}\mathsf{k}^\rho\|\mathcal{Z})$, we may view $\mathcal{D}\mathsf{k}^\pi$ as an adversary attacking π with respect to the environment $\mathcal{Z}'$.

Since π t-WCP-emulates ϕ, there is $\mathcal{S}'_k = \{\mathcal{S}'^H_1, \ldots, \mathcal{S}'^H_n, \mathcal{S}'^H(k), \mathcal{S}'^L\}$ s.t. $EXEC_{\pi,\mathcal{D}\mathsf{k}^\pi,\mathcal{Z}'} \approx EXEC_{\phi,(\mathcal{S}'_k\|\mathcal{D}\mathsf{k}^\pi),\mathcal{Z}'}$, i.e. $EXEC_{\pi,\mathcal{D}\mathsf{k}^\pi,\mathcal{Z}'} \approx EXEC_{\phi,\mathcal{S}'_k,\mathcal{Z}'}$. We take the simulator $\mathcal{S}_k = (\mathcal{S}'_k\|\mathcal{D}\mathsf{k}^\rho)$, where $\mathcal{S}'_k$ and $\mathcal{D}\mathsf{k}^\rho$ are just running in parallel without any interaction. This is a suitable simulator for the protocol $\rho[\phi \mapsto \pi]$, having all necessary ports. Since there is no interaction between $\mathcal{S}'_k$ and $\mathcal{D}\mathsf{k}^\rho$, the construction of $\mathcal{S}_k$ satisfies Definition 5.3.

We want to show that $\mathcal{S}_k$ is a suitable simulator for $\rho[\phi \to \pi]$. Assume by contrary that $\mathcal{S}_k$ is not suitable. That is, $|EXEC_{\rho[\phi\to\pi],\mathcal{D}\mathsf{k},\mathcal{Z}} - EXEC_{\rho,\mathcal{S}_k,\mathcal{Z}}| \geq \varepsilon$. Since $\mathcal{S}_k = (\mathcal{S}'_k\|\mathcal{D}\mathsf{k}^\rho)$ and $\mathcal{Z}' = (\mathcal{D}\mathsf{k}^\rho\|\mathcal{Z})$, we have $EXEC_{\rho,\mathcal{S}_k,\mathcal{Z}} \approx EXEC_{\phi,\mathcal{S}'_k,\mathcal{Z}'}$. Since $\mathcal{D}\mathsf{k} = (\mathcal{D}\mathsf{k}^\pi\|\mathcal{D}\mathsf{k}^\rho)$, we also have $EXEC_{\rho[\phi\to\pi],\mathcal{D}\mathsf{k},\mathcal{Z}} \approx EXEC_{\pi,\mathcal{D}\mathsf{k}^\pi,\mathcal{Z}'}$.

Putting all together, we get $|EXEC_{\pi,\mathcal{D}\mathsf{k}^\pi,\mathcal{Z}'} - EXEC_{\phi,\mathcal{S}'_k,\mathcal{Z}'}| \geq \varepsilon$, which contradicts the assumption that π t-WCP-emulates ϕ. □

The quantities used in the proof are depicted in Figure 5.5.

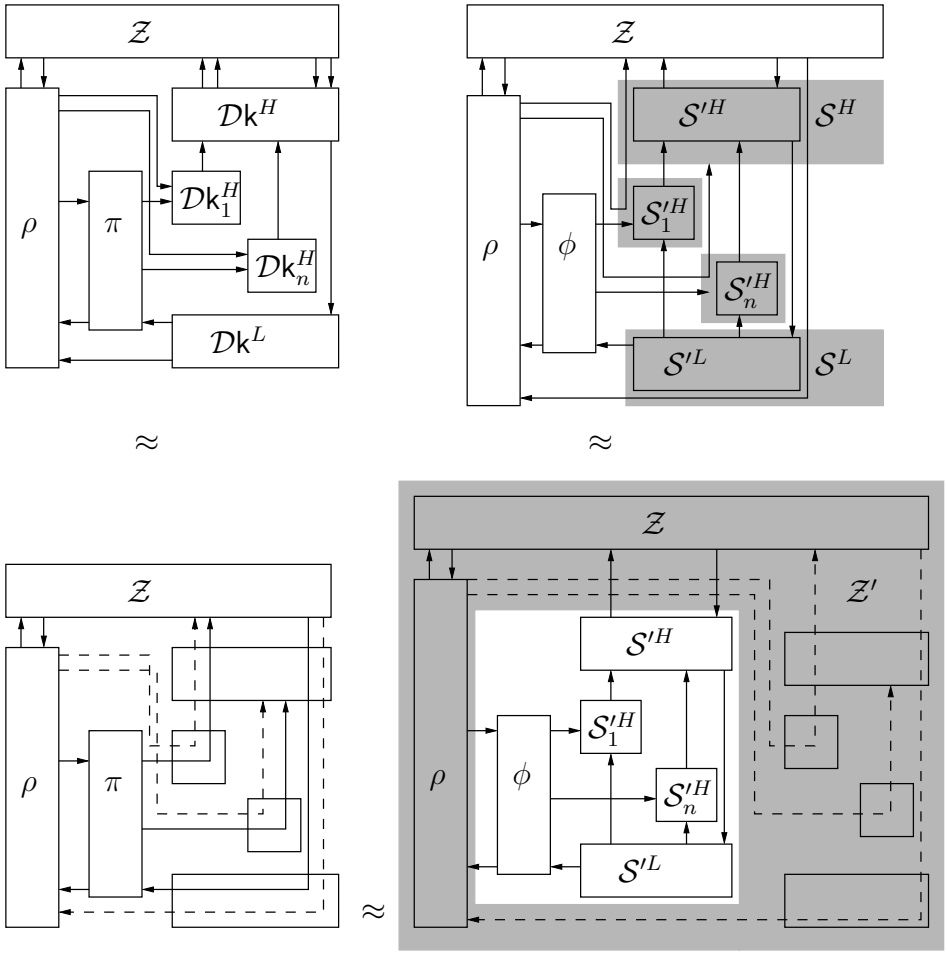


Figure 5.5: WCP composition theorem

5.3.7 Relations to the Existing Notions

We want to show that no attack that the UC model detects remains unnoticed by the WCP model. Namely, we show that t-WCP-emulation implies UC-emulation, and hence our security definition is stronger. Similarly to CP, failure in achieving t-WCP-specific properties provides UC security fallback on the assumption that at most t parties are corrupted in UC.

Transformations of Different Models

Since the ports between $\mathcal{F}$ and $\mathcal{A}_S$ are different in UC and WCP models, we cannot use the same functionality $\mathcal{F}$ in both UC and WCP models at once. We need to define a transformation between UC and WCP functionalities. In this section we describe how an ideal functionality $\mathcal{F}^{WCP}$ defined in WCP model can be mapped to/from the corresponding functionality $\mathcal{F}^{UC}$, $\mathcal{F}^{CP}$, $\mathcal{F}^{LUC}$. We use the notation $\downarrow_Y^X$ for the transformation from a functionality of the model X to the functionality of the model Y.

Let $c(i)$ be the index of the adversary corrupting P_i. Let $\mathcal{C}_k$ be the set of all parties corrupted by $\mathcal{A}_k^H$.

WCP and UC. Let $\mathcal{A}$ be the monolithic UC adversary corrupting at most t parties, and $\{\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^H, \mathcal{A}^L\}$ the t-coalition split adversary. Since in UC the view of *all* corrupted parties goes to $\mathcal{Z}$, the UC adversary $\mathcal{A}$ corresponds to *one* WCP adversary $\mathcal{A}_k^H$, taking into account its collaboration with $\mathcal{A}^L$.

Definition 5.6 (WCP to UC)**.** Let $\mathcal{F}^{WCP}$ be a functionality in the WCP model. The functionality $\mathcal{F}^{UC} = \downarrow_{UC(k)}^{WCP} (\mathcal{F}^{WCP})$ behaves the same as $\mathcal{F}^{WCP}$ with the following differences in the interface:

1. Upon receiving an input (in) from $\mathcal{A}$, $\mathcal{F}^{UC}$ behaves as $\mathcal{F}^{WCP}$ would upon receiving input (in) from $\mathcal{A}^L$.

2. Whenever $\mathcal{F}^{WCP}$ generates an output (out) to the adversary $\mathcal{A}_k^H$, $\mathcal{F}^{UC}$ gives (out) to $\mathcal{A}$. The outputs of $\mathcal{F}^{WCP}$ generated for $\mathcal{A}_j^H$, $j \neq k$ are ignored.

In this way, the transformation $\downarrow_{UC}^{WCP}$ is parametrized by k, and there are up to n different transformations $\downarrow_{UC(1)}^{WCP} (\mathcal{F}^{WCP}), \ldots, \downarrow_{UC(n)}^{WCP} (\mathcal{F}^{WCP})$, depending on which coalition is treated as an adversary in UC model. We note that $k = 1$ provides the strongest security definition, since $k > 1$ assume only one passively corrupted party. Nevertheless, the transformations for $k > 1$ can still be interesting in the cases of asymmetric protocols, where it is known in advance that some

party is unconditionally honest. As we prove in Theorem 5.2, any protocol WCP-emulating $\mathcal{F}^{WCP}$ will also UC-emulate $\mathcal{F}^{UC} =\downarrow^{WCP}_{UC(k)} (\mathcal{F}^{WCP})$ for all $k \in [n]$.

A more general solution would be to allow $\mathcal{A}$ to corrupt parties that are controlled by *different* adversaries $\mathcal{A}^H_k$. If the protocol assumes that any party may be corrupted, and there are no unconditionally honest parties, then this case would as well be covered by $\downarrow^{WCP}_{UC(k)}$, treating these parties as one coalition. If the initial WCP protocol makes assumptions that some particular party is unconditionally honest, then the resulting UC protocol would be insecure, allowing to merge together the views of the corrupted parties and the unconditionally honest party. In UC model, such an honest party would never be treated as corrupted anyway. Hence, without loss of generality, we end up with the transformations $\downarrow^{WCP}_{UC(k)}$.

Definition 5.7 (UC to t-WCP)**.** Let $\mathcal{F}^{UC}$ be a functionality in the UC model. The functionality $\mathcal{F}^{WCP} =\downarrow^{UC}_{WCP} (\mathcal{F}^{UC})$ behaves the same as $\mathcal{F}^{UC}$ with the following differences in the interface:

1. Upon receiving an input (in) from $\mathcal{A}^L$, $\mathcal{F}^{WCP}$ behaves as $\mathcal{F}^{UC}$ upon receiving input (in) from $\mathcal{A}$.

2. Whenever $\mathcal{F}^{UC}$ generates an output (out) to $\mathcal{A}$, $\mathcal{F}^{WCP}$ gives (out) to the adversary $\mathcal{A}^H_1$.

The transformation works in a straightforward way, keeping the description of interaction with the active adversarial coalition, and without specifying anything about the information leaked to the honest parties. This additional specification is in general non-trivial, and it should be done individually for each ideal functionality. We will give some examples in Section 5.4, when we covert the UC functionalities of Chapter 4 to WCP functionalities.

WCP and CP. Similarly to UC, since in CP the view of *all* corrupted parties goes to $\mathcal{Z}$, the CP adversary $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ may corrupt up to t parties to make it comparable with t-WCP. Differently from UC, the parties of CP may be corrupted by different adversaries. Let CP adversary corrupt $t' \leq t$ parties actively, and the remaining $t - t'$ parties passively. We assume that the same parties are actively corrupted in both models.

As the result, our transformation will split the active adversary $\mathcal{A}^H_1$ into $\mathcal{A}_i$ for $i \in \mathcal{C}_1$ when transforming WCP protocols to CP protocols, and it will merge the actively corrupted $\mathcal{A}_i$ to $\mathcal{A}^H_1$ when transforming in the other direction. Otherwise, there will be one-to one correspondence between $\mathcal{A}^H_i$ and $\mathcal{A}_i$.

Definition 5.8 (CP to WCP)**.** Let $\mathcal{F}^{CP}$ be a functionality in the CP model. The functionality $\mathcal{F}^{WCP} =\downarrow^{CP}_{WCP} (\mathcal{F}^{CP})$ behaves the same as $\mathcal{F}^{CP}$ with the following differences in the interface:

1. Upon receiving an input (in, i) from $\mathcal{A}^L$, $\mathcal{F}^{WCP}$ behaves as $\mathcal{F}^{CP}$ would upon receiving input (in) from the adversary $\mathcal{A}_i$.

2. Whenever $\mathcal{F}^{CP}$ generates an output (out) to the adversary $\mathcal{A}_i$, $\mathcal{F}^{WCP}$ gives (out) to $\mathcal{A}^H_{c(i)}$.

Definition 5.9 (WCP to CP)**.** Let $\mathcal{F}^{WCP}$ be a functionality in the WCP model. The functionality $\mathcal{F}^{CP} =\downarrow^{WCP}_{CP} (\mathcal{F}^{WCP})$ behaves the same as $\mathcal{F}^{WCP}$ with the following differences in the interface:

1. Upon receiving an input (in) from some $\mathcal{A}_i$, $\mathcal{F}^{CP}$ behaves as $\mathcal{F}^{WCP}$ upon receiving input (in) from $\mathcal{A}^L$.

2. Whenever $\mathcal{F}^{WCP}$ generates output (out) to $\mathcal{A}^H_k$, $\mathcal{F}^{CP}$ gives (out) to all the adversaries $\mathcal{A}_j$ such that $j \in \mathcal{C}_k$.

WCP and LUC. Similarly to the CP case, we assume that there are t' parties actively corrupted by the LUC adversary, and $t - t'$ passively corrupted. The idea is to first merge all the adversaries $\mathcal{A}_{i,j}$ into one adversary $\mathcal{A}_i$, since $\mathcal{A}_{i,j}$ models the communication between P_i and P_j as seen by P_i. These adversaries are then handled similarly to CP adversaries, merging the active adversaries into $\mathcal{A}^H_1$.

Definition 5.10 (LUC to WCP)**.** Let $\mathcal{F}^{LUC}$ be a functionality in the LUC model. The functionality $\mathcal{F}^{WCP} =\downarrow^{LUC}_{WCP} (\mathcal{F}^{LUC})$ behaves the same as $\mathcal{F}^{LUC}$ with the following differences in the interface:

1. Upon receiving an input $(in, (i, j))$ from $\mathcal{A}^L$, $\mathcal{F}^{WCP}$ behaves as $\mathcal{F}^{LUC}$ would upon receiving input (in) from the adversary $\mathcal{A}_{i,j}$.

2. Whenever $\mathcal{F}^{LUC}$ generates an output (out) to the adversary $\mathcal{A}_{i,j}$, $\mathcal{F}^{WCP}$ gives $(out, (i, j))$ to $\mathcal{A}^H_{c(i)}$.

Definition 5.11 (WCP to LUC)**.** Let $\mathcal{F}^{WCP}$ be a functionality in the WCP model. The functionality $\mathcal{F}^{LUC} =\downarrow^{WCP}_{LUC} (\mathcal{F}^{WCP})$ behaves the same as $\mathcal{F}^{WCP}$ with the following differences in the interface:

1. Upon receiving an input (in) from some $\mathcal{A}_{i,j}$, $\mathcal{F}^{LUC}$ behaves as $\mathcal{F}^{WCP}$ upon receiving input (in) from $\mathcal{A}^L$.

2. Whenever $\mathcal{F}^{WCP}$ generates output (out) to $\mathcal{A}^H_k$, $\mathcal{F}^{LUC}$ gives (out) to $\mathcal{A}_{i,j}$ for all $i \in \mathcal{C}_k$, $j \in [n]$, $j \neq i$.

These transformations can be easily extended to protocols using ideal functionalities as subroutines, similarly to CP and LUC transformations. In particular, if a protocol π^X uses a subroutine $\mathcal{F}^X$ defined in the model X, then replacing $\mathcal{F}^X$ with a new subroutine $\mathcal{F}^Y =\downarrow^X_Y (\mathcal{F}^X)$ gives us a protocol π^Y defined in the model Y.

Relations of Different Models

In Section 5.3.7 we have defined transformations for different models. These transformations modify the definitions of ideal functionalities $\mathcal{F}$ in a way similar to how communication of the computing parties with the adversary is changed between different models. In this section, we prove some strength relationships between WCP and the other models.

We show that, assuming that the number of corrupted parties is the same, then WCP emulation implies UC emulation. As we show in Section 5.3.8, the converse does not hold.

Theorem 5.2. *If the protocol π t-WCP-emulates the protocol ϕ, then $\downarrow^{WCP}_{UC(k)}(\pi)$ UC-emulates $\downarrow^{WCP}_{UC(k)}(\phi)$ for all $k \in [n]$, assuming t corrupted parties.*

Proof. Suppose that $\pi^{WCP} := \pi$ t-WCP emulates $\phi^{WCP} := \phi$. Let $\mathcal{S}_k = \{\mathcal{S}_1^H, \ldots, \mathcal{S}_n^H, \mathcal{S}^H(k), \mathcal{S}^L\}$ be the simulator that exists due to t-WCP emulation for any t-coalition split adversary.

Let $\mathcal{S} = \{\mathcal{S}_k^H, \mathcal{S}^L, \blacksquare\}$, where:

- $\mathcal{S}_k^H$ delivers all messages directly to $\mathcal{A}$ and to $\mathcal{S}^L$ instead of $\mathcal{S}^H(k)$;
- $\mathcal{S}^L$ receives messages directly from $\mathcal{S}_k^H$ and $\mathcal{A}$ instead of $\mathcal{S}^H(k)$, and outputs all messages intended for $\mathcal{S}_i^H$, $i \neq k$ to $\blacksquare$ instead;
- $\blacksquare$ is an ITM that does not output any messages.

We show that it is a suitable simulator for the UC protocols $\pi^{UC} := \downarrow^{WCP}_{UC(k)}(\pi)$ and $\phi^{UC} := \downarrow^{WCP}_{UC(k)}(\phi)$. We prove the theorem for $k = 1$, assuming that there are t parties actively corrupted by the UC adversary. The case $k > 1$ follows directly from the $k = 1$ case, since it considers a weaker UC adversary, so the simulation can be only easier for $k > 1$.

Assume by contradiction that there are $\mathcal{A}$, $\mathcal{Z}$ such that $|EXEC_{\pi^{UC},\mathcal{A},\mathcal{Z}} - EXEC_{\phi^{UC},(\mathcal{S}\|\mathcal{A}),\mathcal{Z}}| \geq \varepsilon$. Define an adversary $\mathcal{A}' = \{\mathcal{A}_1'^H, \ldots, \mathcal{A}_n'^H, \mathcal{A}'^H, \mathcal{A}'^L\}$ attacking π^{WCP}, where $\mathcal{A}_1'^H$ is the true adversary, just forwarding all messages from π^{WCP} to $\mathcal{A}^H$, and $\mathcal{A}^L = \mathcal{A}$, where the messages that $\mathcal{A}$ expects from π^{UC} are coming from $\mathcal{A}^H$. By construction, $\mathcal{A}_1'^H$ forwards all messages of π^{WCP} to $\mathcal{A}^L = \mathcal{A}$. By Definition 5.6, the protocol $\downarrow^{WCP}_{UC}(\pi)$ is defined similarly to π, but all messages that were meant for $\mathcal{A}_1'^H$ are now sent to $\mathcal{A}$. Hence we have $EXEC_{\pi^{UC},\mathcal{A},\mathcal{Z}} = EXEC_{\pi^{WCP},\mathcal{A}',\mathcal{Z}}$.

Let us look at the composition $(\mathcal{S}_{\mathsf{true}(\mathcal{A}')}\|\mathcal{A}') = (\mathcal{S}_1\|\mathcal{A}')$. All messages of $\mathcal{S}_j^H$ that $\mathcal{S}^H$ delivers to $\mathcal{A}_j'^H$ for $j \neq 1$ will be lost since $\mathcal{A}_j'^H$ are false adversaries. Hence the interaction of $\mathcal{S}_1$ with $\mathcal{A}'$ would not change if we connected $\mathcal{S}_1^H$ directly

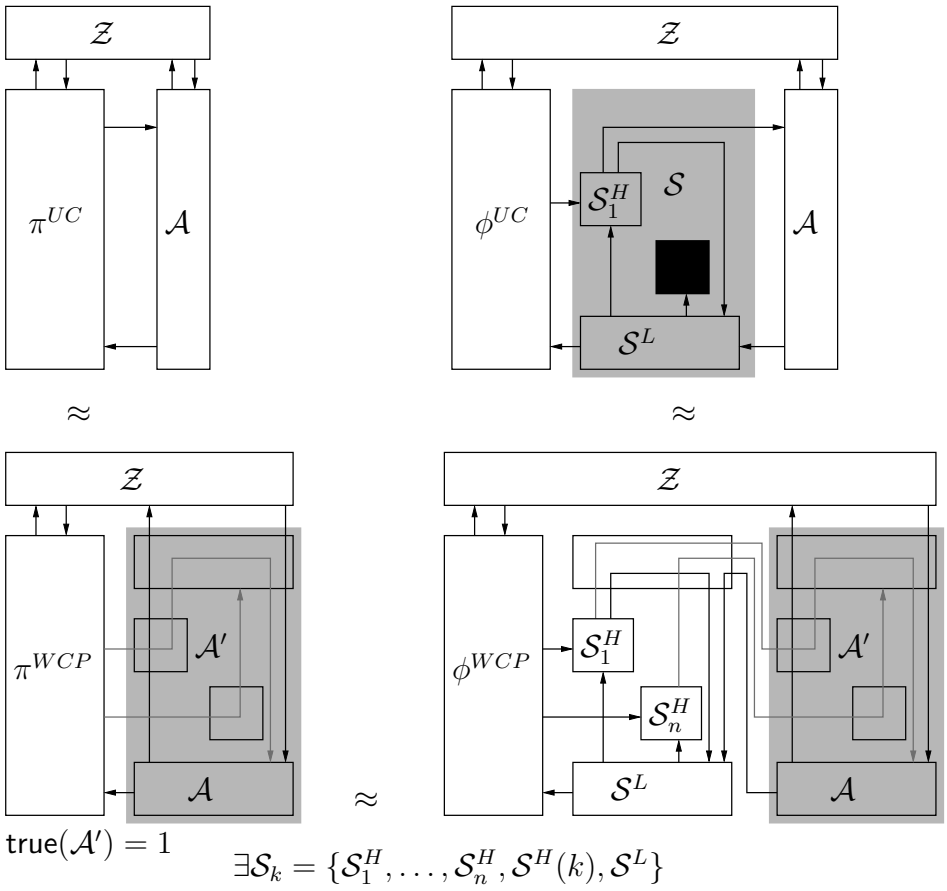


Figure 5.6: WCP emulation implies UC emulation

to $\mathcal{A}_1'^H$ and discarded all $\mathcal{S}_j^H$ for $j \neq 1$. Since $\mathcal{A}_1^H$ is the true adversary, $\mathcal{S}^L$ receives the same messages of $\mathcal{S}_1^H$ as it does in $\mathcal{S}_k$. By discussion similar to π^{UC} case, in ϕ^{UC} exactly those outputs that $\mathcal{A}_1^H$ received from ϕ^{WCP} are transmitted to $(\mathcal{S}\|\mathcal{A})$. We get $EXEC_{\pi^{UC},(\mathcal{S}\|\mathcal{A}),\mathcal{Z}} = EXEC_{\pi^{WCP},(\mathcal{S}_{\mathsf{true}(\mathcal{A}')}\|\mathcal{A}'),\mathcal{Z}}$.

We get that, if $|EXEC_{\pi^{UC},\mathcal{A},\mathcal{Z}} - EXEC_{\phi^{UC},(\mathcal{S}\|\mathcal{A}),\mathcal{Z}}| \geq \varepsilon$ holds, then also $|EXEC_{\pi^{WCP},\mathcal{A}',\mathcal{Z}} - EXEC_{\phi^{WCP},(\mathcal{S}_{\mathsf{true}(\mathcal{A}')}\|\mathcal{A}'),\mathcal{Z}}| \geq \varepsilon$ holds, contradicting the t-WCP emulation assumption. □

The quantities used in the proof are depicted in Figure 5.6. Analogously, one can prove that $\underline{\mathcal{G}}$-WCP emulation implies $\underline{\mathcal{G}}$-EUC emulation.

We would also like to compare WCP and CP. In general, CP security is stronger. Firstly, it requires a separate simulator $\mathcal{S}_i$ for each party, while WCP uses a single simulator $\mathcal{S}_1^H$ for all actively corrupted parties. Secondly, if the t parties that are

corrupted in CP model are corrupted by different adversaries $\mathcal{A}_j^H$ in the WCP model, then the outputs of *all* adversaries $\mathcal{A}'_i$ will reach $\mathcal{Z}$, while only one $\mathcal{A}_j^H$ delivers its outputs to $\mathcal{Z}$ in WCP.

We show that, assuming that the number of corrupted parties is the same, then CP emulation implies WCP emulation. As we show in Section 5.3.8, the converse does not hold.

Theorem 5.3. *If the protocol π CP-emulates a protocol ϕ assuming t corrupted parties, then $\downarrow_{WCP}^{CP}(\pi)$ t-WCP emulates $\downarrow_{WCP}^{CP}(\phi)$.*

Proof. Let $\pi^{CP} := \pi$, $\phi^{CP} := \phi$, $\pi^{WCP} := \downarrow_{WCP}^{CP}(\pi)$, $\phi^{WCP} := \downarrow_{WCP}^{CP}(\phi)$. The CP emulation gives a simulator $\mathcal{S}' = \{\mathcal{S}'_1, \ldots, \mathcal{S}'_n\}$ such that, for any $\mathcal{Z}$ and for any CP adversary $\mathcal{A}'$, we have $EXEC_{\pi^{CP},\mathcal{A}',\mathcal{Z}} \approx EXEC_{\phi^{CP},(\mathcal{S}'\|\mathcal{A}'),\mathcal{Z}}$.

Let $\mathcal{C}$ denote the set of actively corrupted parties. Take $\mathcal{S}_1^H = \{\mathcal{S}'_i \mid i \in \mathcal{C}\}$, and $\mathcal{S}_i^H = \mathcal{S}'_i$ for $i \notin \mathcal{C}$. Let $\mathcal{S}^L$ contain $\mathcal{S}'_j$ for all $j \in [n]$, and let it forward messages from $\mathcal{S}^H(k)$ to ϕ^{WCP} and to $\mathcal{S}_j^H$. Define $\mathcal{S}^H(k)$ in a standard way. Let $\mathcal{S}_k = \{\mathcal{S}_1^H, \ldots, \mathcal{S}_n^H, \mathcal{S}^H(k), \mathcal{S}^L\}$.

Suppose that this choice of $\mathcal{S}_k$ is not good, and we have found an adversary $\mathcal{A} = \{\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^H, \mathcal{A}^L\}$ and $\mathcal{Z}$ that can break t-WCP, i.e. $|EXEC_{\pi^{WCP},\mathcal{A},\mathcal{Z}} - EXEC_{\phi^{WCP},(\mathcal{S}_{\mathsf{true}(\mathcal{A})}\|\mathcal{A}),\mathcal{Z}}| \geq \varepsilon$. Let a CP adversary $\mathcal{A}'$ be defined as $\mathcal{A}'_i = (\mathcal{A}_1^H\|\mathcal{A}^L)$ for all $i \in \mathcal{C}$, and $\mathcal{A}'_i = (\mathcal{A}_j^H\|\mathcal{A}^L)$ for some $j \neq i$ for all $i \notin \mathcal{C}$. Let $\mathcal{Z}' = (\mathcal{A}^H\|\mathcal{Z})$. In other words, we have taken $\mathcal{A}'$ such that $(\mathcal{A}'\|\mathcal{Z}') = (\mathcal{A}\|\mathcal{Z})$, and just partitioned it logically into $\mathcal{A}'_i$, giving each $\mathcal{A}'_i$ its own copy of $\mathcal{A}^L$. Let $j = \mathsf{true}(\mathcal{A})$. We show that, if $\mathcal{A}'$ corrupts the parties of $\mathcal{C}_j$, then $\mathcal{A}'$ and $\mathcal{Z}'$ break CP emulation.

- Since $\mathcal{S}^L$ may deliver everything that it gets from $\mathcal{A}^L$ to $\mathcal{S}_j^H$, and $\mathcal{S}_j^H$ contains the code of $\mathcal{S}'_i$ for all $i \in \mathcal{C}_j$, it is able to simulate the messages leaving $\mathcal{S}_j^H$ and entering $\mathcal{A}_j^H$ in such a way that they come from the same distribution as the messages exiting $\mathcal{S}'_i$ and entering $\mathcal{A}'_i$ in the CP model for all $i \in \mathcal{C}_j$.

- Conversely, $\mathcal{S}_j^H$ is able to deliver arbitrary messages to $\mathcal{S}^L$ via $\mathcal{S}^H(j)$. Hence $\mathcal{S}^L$ is able to simulate the messages leaving $\mathcal{S}^L$ and entering ϕ^{WCP} in such a way that they come from the same distribution as the messages exiting $\mathcal{S}'_i$ and entering ϕ^{CP} for all $i \in \mathcal{C}_j$.

As a summary, for $j = \mathsf{true}(\mathcal{A})$, we get the following. By construction of $\mathcal{A}'$, since $(\mathcal{A}'\|\mathcal{Z}') = (\mathcal{A}\|\mathcal{Z})$, and their interaction with the protocol is the same, we have $EXEC_{\pi^{CP},\mathcal{A}',\mathcal{Z}'} \approx EXEC_{\pi^{WCP},\mathcal{A},\mathcal{Z}}$. Similarly, the simulators $\mathcal{S}'_i$ for $i \in \mathcal{C}_j$, taken altogether, interact with ϕ^{CP} and $\mathcal{A}'_i$ exactly in the same way as $\mathcal{S}_j^H$

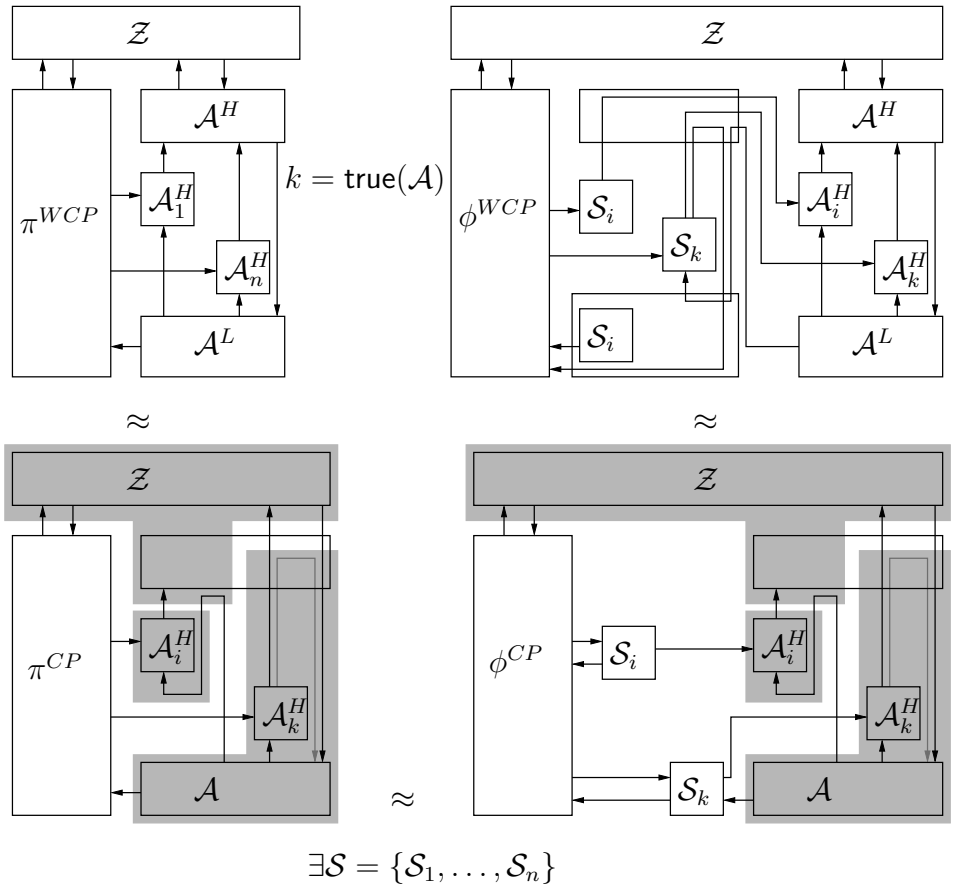


Figure 5.7: CP emulation implies WCP emulation

and $\mathcal{S}^L$ together interact with ϕ^{WCP} and $\mathcal{A}_j^H$, $\mathcal{A}^L$, so $EXEC_{\pi^{CP},(\mathcal{S}'\|\mathcal{A}'),\mathcal{Z}'} \approx EXEC_{\pi^{WCP},(\mathcal{S}\|\mathcal{A}),\mathcal{Z}}$. Hence $|EXEC_{\pi^{CP},\mathcal{A}',\mathcal{Z}'} \approx EXEC_{\phi^{CP},(\mathcal{S}'\|\mathcal{A}'),\mathcal{Z}'}| \geq \varepsilon$, which contradicts the assumption that π^{CP} CP-emulates ϕ^{CP} □

The quantities used in the proof are depicted in Figure 5.7. Since the LUC model is a generalization of the CP model, and the adversaries $\mathcal{A}_{i,1}, \ldots, \mathcal{A}_{i,n}$ could be treated as a single adversary getting all the messages received by $\mathcal{A}_i$, we conclude that WCP is also weaker than LUC.

Corollary 5.1. *If the protocol π LUC-emulates a protocol ϕ assuming t corrupted parties, then $\downarrow_{WCP}^{LUC}(\pi)$ t-WCP emulates $\downarrow_{WCP}^{LUC}(\phi)$.*

5.3.8 Applicability of the WCP Model

In this section we show why WCP is a suitable model for pointing out the attacks we mentioned in Section 5.2. We present some properties related to leaking information to an honest party that can be captured by t-WCP, but not by UC, CP, LUC. Since CP lets the adversaries to communicate through an arbitrary resource R, the security in CP model depends on the particular choice of R, which allows it to be stronger as well as weaker than the other models. In order to make the definitions similar, we assume that R delivers to $\mathcal{A}_i$ the internal state of P_i, and the adversary $\mathcal{A}_i$ may also replace any message m sent by P_i by a message m^* of $\mathcal{A}_i$'s own choice.

The relations of our protocols and functionalities with the adversaries are described as $\mathcal{A}(i)$, where i is some party identifier, and $\mathcal{A}(i)$ corresponds to *all i-related adversaries*, which is just $\mathcal{A}$ for UC, $\mathcal{A}_i$ for CP, $\mathcal{A}_{c(i)}$ for WCP, and $\mathcal{A}_{i,1}, \ldots, \mathcal{A}_{i,n}$ for LUC. In fact, we define the initial functionality in CP model, transforming it to UC, WCP, and LUC. More details about transformations of ideal functionalities between different models can be found in Section 5.3.7.

Bad Key Attacks

First of all, we present an ideal functionality $\mathcal{F}_0$ and two of its possible realizations π_1 and π_2. We show that, while for UC, CP, LUC these realizations either both realize or do not realize $\mathcal{F}_0$, they are different in t-WCP model. Let $Enc(k, m)$ be a pseudorandom permutation [45, Definition 3.7.2], i.e. a function such that, if k is sampled from a uniform distribution K, then the distribution of $y = Enc(K, m)$ is computationally indistinguishable from a uniform distribution, for any fixed m.

Ideal. The ideal functionality $\mathcal{F}_0$ takes a secret s from a certain party P_i. If P_i is actively corrupted, then $\mathcal{F}_0$ outputs s to each $\mathcal{A}(j)$ for $j \in [n]$. The adversary is allowed to abort the protocol. If it does not, $\mathcal{F}_0$ outputs 0 to each party.

Protocol 1. Consider the protocol π_1 where a (symmetric) key is generated as $k = \sum_{\ell \in \mathcal{I}} k_\ell$ where $\mathcal{I}$ is a set of arbitrarily chosen t parties that are supposed to generate k_ℓ from uniform distribution. All k_ℓ are sent to the party P_i that encrypts a secret s with this key and sends $Enc(k, s)$ to some party P_j. If any party refuses to send its message, the protocol aborts.

Protocol 2. Consider an analogous protocol π_2 which works in exactly the same way, but where P_i itself generates one more share k_{t+1} of k, and sends it to all other parties.

We now compare these protocols in various models.

- **UC** Assuming that the total number of corrupted parties is at most t, both π_1 and π_2 UC-realize $\mathcal{F}_0$. If P_i is corrupted, then S gets s from $\mathcal{F}_0$ and can simulate everything. Otherwise, the adversary either gets only the key k (if

P_j is not corrupted), or it gets $Enc(k, s)$ and up to all shares of k except one (if P_j is corrupted). If the number of corrupted parties is at least $t + 1$, then both protocols are insecure since all the shares of the key and the $Enc(k, s)$ may leak to $\mathcal{Z}$.

- **CP, LUC** If P_i is corrupted, then the key generating parties may use their shares of k as side channels for collaborating with $\mathcal{A}(i)$, and hence neither π_1 nor π_2 does not realize $\mathcal{F}_0$. Let P_i be honest. Assuming that the total number of corrupted parties is at most t, the functionalities π_1 and π_2 both realize $\mathcal{F}_0$. If at least one key generating party is honest, the simulator $S(j)$ only needs to simulate $Enc(k, s)$ as if the key was uniform. If all the key generating parties are corrupt, then k might not be uniform, but in this case P_j is uncorrupted, and S_j does not have to simulate anything. If the total number of corrupted parties is at least $t + 1$, then both the k and $Enc(k, s)$ may leak to $\mathcal{Z}$, and hence π_1 and π_2 are both insecure, similarly to UC.

- **WCP** The protocol π_2 does t-WCP-realize $\mathcal{F}_0$, but π_1 does not. If P_i is corrupted, then all S_j^H get s from $\mathcal{F}_0$, and S^L gets from $\mathcal{A}^L$ all the shares of k that S^L delivers to all S_j^H, so these side-channels are not taken into account by WCP. Let P_i be honest. In π_1, if all the t key generating parties are corrupted, then S_j^H has to simulate $Enc(k, s)$ based on the bad key k that no longer comes from uniform distribution and might be known by $\mathcal{Z}$. Although S^L might have sent the bad key k to S_j^H, it still does not know s, and hence cannot simulate $Enc(k, s)$. In π_2, the key k comes from a uniform distribution in any case, since at least one share is generated by the uncorrupted P_i itself. The question is whether k may leak to $\mathcal{Z}$ if all the key generating parties are controlled by an adversarial coalition of size t, as they also get the final share k_{t+1} at some moment. We care about the simulation by S_j^H only if $\mathcal{A}_j^H$ is the true adversary. In this case, the entire key generating coalition has been controlled by a false adversary that never leaks the final share k_{t+1} to $\mathcal{Z}$.

We analyze a particular multiparty computation protocol of [80] related to bad key generation. This is a 3-party protocol with one malicious party, where the parties P_1, P_2, P_3 compute some function f on input bits x_1, x_2, x_3^*. The party P_3 shares its input as $x_3^* = x_3 \oplus x_4$, and sends x_3 to P_1 and x_4 to P_2. The parties P_1 and P_2 agree on a common randomness r. They use r to construct a garbled circuit (GC) F that computes $f'(x_1, x_2, x_3, x_4) = f(x_1, x_2, x_3^*)$, and use oblivious transfer (OT) to deliver the bits (x_1, x_3) and (x_2, x_4) to P_3. For our security analysis, the details of GC and OT are not important, and it is only essential that the security of the inputs x_1 and x_2 depends on the randomness r. Hence, without loss of generality, we assume that P_1 computes the encryption

$X_1 = Enc(r, (x_1, x_3))$, P_3 computes the encryption $X_2 = Enc(r, (x_2, x_4))$, and they send X_1, X_2 to P_3 who does some computation with these values, obtaining $f(x_1, x_2, x_3^*)$. As far as P_3 does not know r, it cannot infer x_1 and x_2 from X_1 and X_2.

The authors of [80] mention that the security would be indeed broken if a malicious P_1 sends r to P_3, allowing it to extract x_2 from X_2. This attack is not covered by UC. Indeed, it is reasonable to assume that an honest P_3 will not communicate with P_1 using any side-channels. This attack would not be noticed also by WCP, since it does not take into account side-channel attacks. However, P_1 may perform this attack quietly, without using side-channels, depending on how r is generated.

Suppose that the initial randomness r is generated by P_1 alone, and P_1 just delivers this r to P_2. In UC model, P_1 has no reason to choose a bad r since it does not help P_1 to gain any information anyway. Moreover, a bad r may result in leaking P_1's own secret to P_3. However, P_1 may still sacrifice its own input secrecy and intentionally choose r that has low entropy. If we look at the view of P_3 (that is not covered by UC if P_1 is corrupted), we see that it contains x_2. As the result, the protocol provides a completely legitimate way of opening x_2 to P_3.

This attack is detected in 1-WCP model. Let P_1 be corrupted by an active $\mathcal{A}_1^H$, and P_3 corrupted by a passive $\mathcal{A}_3^H$. Suppose that $\mathcal{A}_3^H$ is the true adversary that just forwards all its data to $\mathcal{Z}$. At some moment, $\mathcal{A}^L$ chooses a low-entropy randomness r that will be delivered by P_1 to P_2. Let $\mathcal{Z}$ be the environment expecting that, in the real protocol execution, the view of $\mathcal{A}_3^H$ will contain the X_2 which is related to x_2 in a certain way defined by the choice of r; i.e. it is waiting for $X_2 \in A(x_2, x_4)$ for a small set $A(x_2, x_4)$ of possible encryptions that depends on x_2 and x_4. This means that $\mathcal{S}_3^H$ has to simulate X_2 that indeed comes from $A(x_2, x_4)$, but it does not know x_2 by default. The ideal functionality outputs only x_3^* to $\mathcal{A}_3^H$, and $\mathcal{S}^L$ may additionally open r to $\mathcal{S}_i^H$. However, it does not help to simulate an element of $A(x_2, x_4)$ without knowing x_2.

At the same time, if we assume that P_1 and P_2 mutually generate a good randomness r running some secure protocol (e.g. using commitment-based coin toss), then the distribution of r does not depend on the inputs chosen by $\mathcal{A}^L$. The simulator $\mathcal{S}_3^H$ should again simulate $Enc(r, (x_2, x_4))$. It generates X_2 according to the *distribution* of r (and not the particular r that was received by $\mathcal{S}_1^H$ and $\mathcal{S}_2^H$). Although in general $X_2 \neq Enc(r, (x_2, x_4))$ for the value r that was actually received by $\mathcal{S}_1^H$ and $\mathcal{S}_2^H$, the definition of WCP ensures that only one of the views of $\mathcal{A}_1^H$, $\mathcal{A}_2^H$ or $\mathcal{A}_3^H$ reaches $\mathcal{Z}$ and $\mathcal{A}^L$. Hence this inconsistency will not be noticed by $\mathcal{Z}$, as X_2 still comes from the distribution it expects.

Bad Sharing Attacks

This attack is somewhat similar to the bad key attack of Section 5.3.8. However, now the shared value itself remains the same, but it will be shared in a bad way, such that a subset of parties smaller than the official threshold will be able to reconstruct the secret. This attack would be more interesting if there were several larger adversarial coalitions. We present its particular case, where a secret gets leaked entirely to some honest party.

Ideal. We take the same ideal functionality $\mathcal{F}_0$ of Section 5.3.8.

Protocol 1. In the protocol π_1, a subset $\mathcal{I}$ of t parties and a subset $\mathcal{J}$ of $(t+1)$ parties ($i \notin \mathcal{I}, \mathcal{I} \cap \mathcal{J} = \emptyset$) are fixed. First, each P_j for $j \in \mathcal{I}$ sends a share s_j to P_i. P_i just generates the last $(t+1)$-th share in such a way that the result would be s, and distributes these shares among the $(t+1)$ parties of $\mathcal{J}$. If any party refuses to send its message, the protocol aborts.

Protocol 2. The protocol π_2 is analogous to π_1 with the only difference that this time P_i generates all the shares $\{s_\ell |\ \ell \in \mathcal{I}\}$ by itself from uniform distribution.

- **UC** Assuming that the total number of corrupted parties is at most t, both π_1 and π_2 UC-realize $\mathcal{F}_0$. If P_i is corrupted, then S gets s from $\mathcal{F}_0$ and can simulate everything. Otherwise, the adversary may get up to t shares of s. If the number of corrupted parties is at least $t+1$, then both protocols are insecure since all the shares may leak.

- **WCP** The protocol π_2 does t-WCP-realize $\mathcal{F}_1$, but π_1 does not. In π_1, the adversary may set up to t shares to a value 0, so that the remaining share will be exactly the secret s that now leaks to some honest party that has not known s yet. At the same time, in π_2 an honest P_i generates all the shares from uniform distribution, and each simulator needs to simulate at most t shares that are distributed uniformly.

- **CP, LUC** Both π_1 and π_2 realize $\mathcal{F}_1$ with at most t parties. Similarly to t-WCP, in π_1 the adversary $\mathcal{A}(i)$ may use the bad sharing as a subliminal channel to leak s to some other party, but if $\mathcal{A}(i)$ falsifies some k shares, there are at most $t-k$ corrupted parties left to receive the other shares, and hence t corruptions are not sufficient for the attack. If the number of parties is at least $(t+1)$, then both π_1 and π_2 are insecure since even if no shares are falsified, in both protocols it may happen that the $t+1$ corrupted parties hold all the $(t+1)$ shares of s.

Our protocol transformations of Chapter 4 are vulnerable to attacks related to bad sharing. In Section 5.4, we will discuss these problems in more details, and we provide standard methods that provide protection against such attacks.

5.4 Protocol Transformations for Achieving the WCP Security

Similarly to Chapter 4, we start from a protocol that is secure against $t < n/2$ passively corrupted parties. In this section, we show how such a protocol can be made secure against $t < n/2$ actively corrupted parties, allowing up to all the other parties to be passively corrupted (i.e. "semihonest majority" assumption).

For any ideal functionality $\mathcal{F}$ or a real protocol Π defined in UC model, we could use the transformations $\downarrow^{UC}_{WCP}(\mathcal{F})$ and $\downarrow^{UC}_{WCP}(\Pi)$ to get the corresponding functionalities and protocols in WCP model. However, the resulting functionality is not necessarily secure in WCP model. The problem is that the transformation only describes the relations between $\mathcal{A}^H_1$, since $\mathcal{F}$ just does not specify which messages are allowed to be sent to the passive adversaries $\mathcal{A}^H_i$, and $\mathcal{S}^H_i$ may fail to simulate the views to passive adversaries $\mathcal{A}^H_i$ in the real protocol. In general, this information cannot be extracted from $\mathcal{F}$, and we need to additionally define which messages can be leaked to honest parties.

In all ideal functionalities of Chapter 4, a message is delivered to $\mathcal{A}$ in the cases when $k \in \mathcal{C}$ for a particular party index k. We turn any UC functionality $\mathcal{F}$ of Chapter 4 to a corresponding WCP functionality $\mathcal{F}^*$ in such a way that, each time when a message m is delivered to $\mathcal{A}$ due to condition $k \in \mathcal{C}$, we deliver m to $\mathcal{A}^H_{c(k)}$ for all $k \in [n]$, where $c(k)$ is the index of the adversary that corrupts P_k. For the protocols, we could define $\Pi^* := \downarrow^{UC}_{WCP}(\Pi)$ in a straightforward way, since the behaviour of parties depends on the adversaries corrupting them, and we do not need to define it manually. However, since almost all protocols of Chapter 4 change, and we want to be more clear with the definitions, we will write out the protocols Π^* in details.

5.4.1 Passive Adversaries

First of all, we show that UC and WCP emulations are equivalent definitions if the adversary is passive. This shows that there is no need to define a special transformation for making a protocol passively secure in WCP model.

Theorem 5.4. *Let π be a protocol that UC-emulates a protocol ϕ in presence of t corrupted parties. Then $\downarrow^{UC}_{WCP}(\pi)$ passively t-WCP emulates $\downarrow^{UC}_{WCP}(\phi)$.*

Proof. Let $\pi^{UC} := \pi$, $\phi^{UC} := \phi$, $\pi^{WCP} := \downarrow^{UC}_{WCP}(\pi)$, $\phi^{WCP} := \downarrow^{UC}_{WCP}(\phi)$. The proof is based on the fact that a passive adversary will not control the corrupted parties, and hence the messages going from the adversary to the protocol do not need to be simulated. This allows to use the UC simulator in place of $\mathcal{S}^H_i$.

Let S' be the simulator that translates the messages between $\mathcal{A}$ and ϕ to simulate π. We show that a suitable choice for $\mathcal{S} = \{\mathcal{S}^H_1, \ldots, \mathcal{S}^H_n, \mathcal{S}^H, \mathcal{S}^L\}$ for

t-WCP emulation is $\mathcal{S}_i^H = \mathcal{S}'$ for all i, and $\mathcal{S}^L$ just forwarding messages from $\mathcal{S}^H$ to $\mathcal{S}_i^H$ and to ϕ^{WCP}.

Suppose that $\mathcal{S}$ is a bad choice, and there exist $\mathcal{A} = \{\mathcal{A}_1^H, \ldots, \mathcal{A}_n^H, \mathcal{A}^H, \mathcal{A}^L\}$ and $\mathcal{Z}$ such that $|EXEC_{\pi^{WCP},\mathcal{A},\mathcal{Z}} - EXEC_{\phi^{WCP},(\mathcal{S}\|\mathcal{A}),\mathcal{Z}}| \geq \varepsilon$. Let $k = \mathsf{true}(\mathcal{A})$. We show that, $\mathcal{A}' = \mathcal{A}$, which can be viewed as an UC adversary after disconnecting $\mathcal{A}_i^H$ for $i \neq k$ from the protocol, will break UC security.

We prove it first for $k = 1$, since $\downarrow_{WCP}^{UC}$ does not specify the leakage to the other adversaries. In this case, the triple $(\mathcal{S}^L, \mathcal{S}_k^H, \mathcal{S}^H)$ is as powerful as $\mathcal{S}'$ is, since they both interact with the same ports of the adversary $\mathcal{A}' = \mathcal{A}$, the same parties belonging to the set $\mathcal{C}$, and get the same inputs from ideal functionalities $\mathcal{F}$ and $\downarrow_{WCP}^{UC}(\mathcal{F})$ respectively. We get $EXEC_{\pi^{UC},\mathcal{A},\mathcal{Z}} \approx EXEC_{\pi^{WCP},\mathcal{A}',\mathcal{Z}'}$ and $EXEC_{\phi^{UC},(\mathcal{S}\|\mathcal{A}),\mathcal{Z}} \approx EXEC_{\phi^{WCP},(\mathcal{S}'\|\mathcal{A}'),\mathcal{Z}'}$, so $|EXEC_{\pi^{WCP},\mathcal{A}',\mathcal{Z}'} - EXEC_{\phi^{WCP},(S'\|\mathcal{A}'),\mathcal{Z}'}| \geq \varepsilon$, contradicting the assumption that π^{UC} UC-emulates ϕ^{UC}. This holds for *any* adversary, not only passive.

We now use the passive adversary assumption to extend the proof to $k > 1$. We take the adversary $\mathcal{A}$ such that $\mathcal{A}$ breaks t-WCP and $\mathsf{true}(\mathcal{A}) = k$, and transform it to an adversary $\mathcal{A}''$ such that $\mathcal{A}''$ also breaks t-WCP and $\mathsf{true}(\mathcal{A}) = 1$. In particular, we just swap the ITMs $\mathcal{A}_1^H$ and $\mathcal{A}_k^H$ of $\mathcal{A}$. Since the adversary is passive, π^{WCP} and ϕ^{WCP} do not expect any inputs from it anyway, and so there is no difference whether $\mathcal{A}^L$ [and hence $\mathcal{S}^L$] outputs anything to the protocol or not. Hence it is not a problem that $\mathcal{S}$ cannot send messages to the parties of $\mathcal{C}_k$ for $k \neq 1$, as $\mathcal{S}'$ is able to do, so $\mathcal{S}$ is still a suitable simulator against $\mathcal{A}''$. Since the adversary with index 1 may corrupt more parties, the resulting adversary $\mathcal{A}''$ may be only stronger. □

The quantities used in the proof are depicted in Figure 5.8. Theorem 5.4 allows us to transfer UC protocols directly to WCP protocols. In general, if $\mathcal{F}^{UC}$ is an ideal functionality of UC model, then the WCP functionality $\mathcal{F}^{WCP} = \downarrow_{WCP}^{UC}(\mathcal{F})$ does not specify the information that may be leaked to honest parties. Proving that $\pi^{WCP} = \downarrow_{WCP}^{UC}(\pi)$ WCP-realizes $\mathcal{F}^{WCP}$ implies that π^{WCP} also WCP-realizes $\mathcal{F}^{WCP}$ that is extended with additional specifications related to honest parties, since the emulation is only easier with these additional specifications that give more information to the simulator.

5.4.2 Fail-Stop Adversaries

A fail-stop adversary [42] follows the protocol as the honest parties do, but it also may force the corrupted parties to abort the protocol. In this case, the protocol may still be secure in t-WCP model, if no one attempts to stop it. However, an attempt to abort the protocol may give the parties some knowledge about the potential set of cheaters. The countermeasures may explicitly require to leak a secret to some honest party.

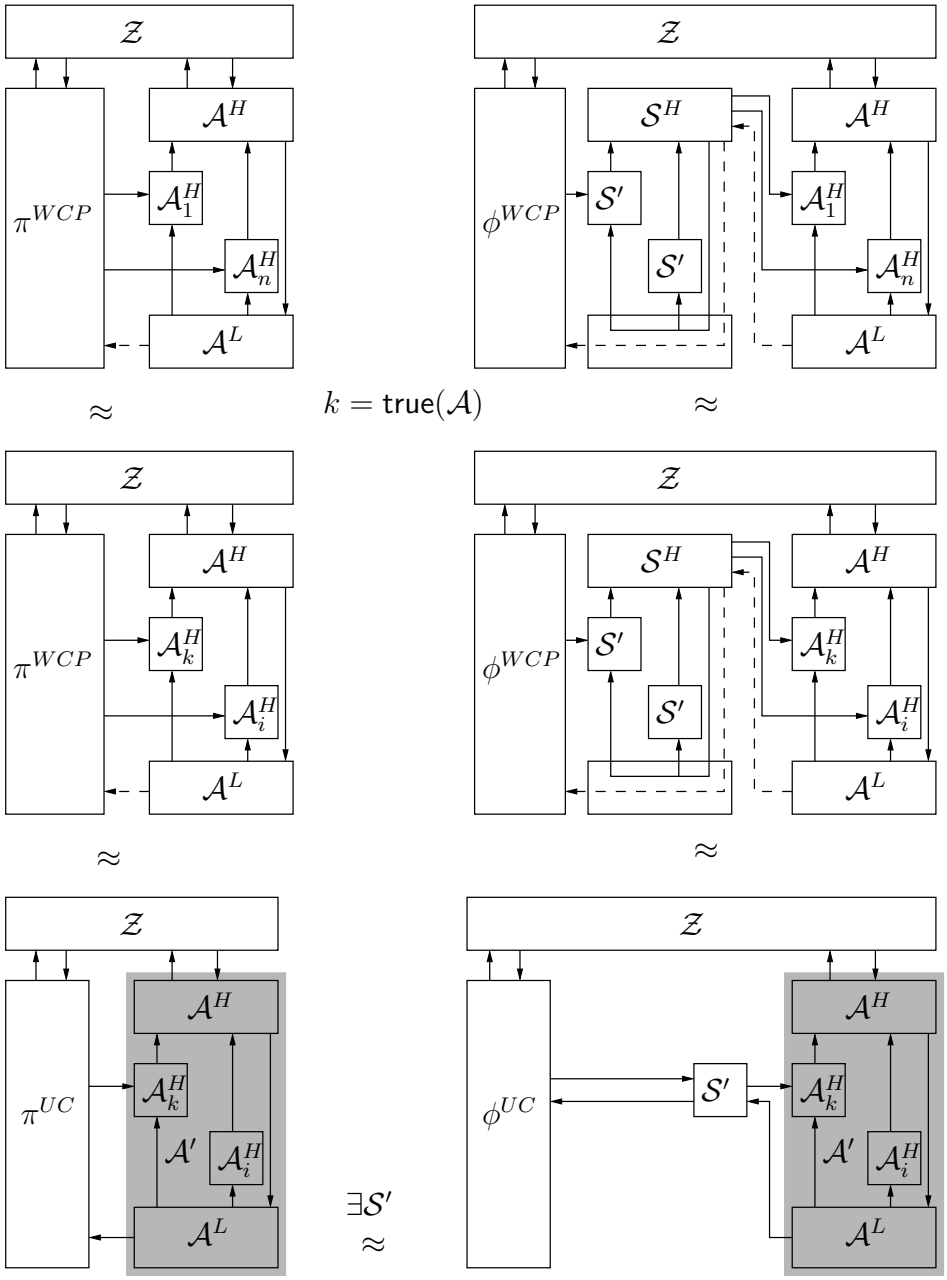


Figure 5.8: UC $\approx$ WCP for a passive adversary

In this section, we will do some t-WCP proofs for particular functionalities. As discussed in Section 5.3.3, we need to describe only the work of one simulator $\mathcal{S}_i$ that receives messages from $\mathcal{A}^L$, sends messages to $\mathcal{A}_i^H$, and communicates with the protocol in both directions. The proof must hold for all $i \in [n]$.

The functionality $\mathcal{F}_{transmit}$ of Figure 4.11 allows communication that is secure against a fail-stop adversary in the ordinary UC model. If we convert it to a WCP functionality, letting each honest party P_i deliver its internal state to the passive adversary $\mathcal{A}_{c(i)}^H$ controlling it, then we get the functionality ${\mathcal{F}_{transmit}}^*$ depicted in Figure 5.9. Compared to $\mathcal{F}_{transmit}$, there is a small modification in the definition of all types of transmissions (including broadcasts): now the messages are revealed to the corresponding adversaries even if the transmission fails and $(\mathsf{cheater}, s(id))$ is output to all parties. It would make no difference for the UC model, since it would just give $\mathcal{A}_S$ the ability to send a message to itself. However, in the WCP model, this does make a difference, since even if the transmission fails, ${\mathcal{A}_S}^L$ may still use it to legally transmit some data to an honest receiver.

As we discussed in Section 5.1, if we directly take the protocol $\Pi_{transmit}$ of Figure 4.12, then $\downarrow_{WCP}^{UC}(\Pi_{transmit})$ will not WCP-realize ${\mathcal{F}_{transmit}}^*$. The reason is, that if there is a conflict between the sender and the receiver, then we cannot let the other parties help in the delivery of the message m, since the inner state of an honest party P_i will be output to $\mathcal{A}_{c(i)}^H$, and $\mathcal{S}_{c(i)}$ will be unable to simulate m. In this subsection, we define a new protocol ${\Pi_{transmit}}^*$ realizing ${\mathcal{F}_{transmit}}^*$ in WCP model.

Cheap mode of $\mathcal{F}_{transmit}$. In order to distinguish better between different protocols, let ${\mathcal{F}_{transmit}^{cheap}}^*$ denote the functionality that works similarly to ${\mathcal{F}_{transmit}}^*$, but only in its cheap mode, outputting $(id, \perp)$ for the corresponding message identifier id at any time when expensive mode should be entered. We claim that ${\Pi_{transmit}}^*$ WCP-implements ${\mathcal{F}_{transmit}^{cheap}}^*$.

- During transmissions and forwardings, as far as there are no conflicts between the sender and the receiver, all the messages are delivered, and no additional information is required to be leaked to honest parties.
- The broadcast and the revealing do not involve any additional messages except m that should be broadcast / revealed to all parties anyway, according to the ideal functionality rules.

Proposition 5.2 states more formally that ${\Pi_{transmit}^{cheap}}^*$ WCP-realizes ${\mathcal{F}_{transmit}^{cheap}}^*$. Availability of ${\Pi_{transmit}^{cheap}}^*$ allows to use ${\mathcal{F}_{transmit}^{cheap}}^*$ as a part of preprocessing that lets us build more complex protocols, including WCP-realization of *expensive* mode of ${\mathcal{F}_{transmit}}^*$.

$\mathcal{F}_{transmit}{}^*$ works with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$, and a party $f(id) \in [n]$ to whom the message should be forwarded by the receiver (if no forwarding is foreseen then $f(id) = r(id)$; for broadcasts the values of $r(id)$ and $f(id)$ do not matter).

• **Initialization:** On input $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$ from all (honest) parties, where $\hat{s}$,$\hat{r}$,$\hat{f}$ are mappings s.t $\mathsf{Dom}(\hat{s}) = \mathsf{Dom}(\hat{r}) = \mathsf{Dom}(\hat{f})$, assign $s \leftarrow \hat{s}$, $r \leftarrow \hat{r}$, $f \leftarrow \hat{f}$. Deliver $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$ to all adversaries ${\mathcal{A}_S}_i^H$.

• **Secure transmit:** On input $(\mathsf{transmit}, id, m)$ from $P_{s(id)}$ and $(\mathsf{transmit}, id)$ from all (honest) parties:

1. For $s(id) \in \mathcal{C}$, let m be chosen by ${\mathcal{A}_S}^L$.
2. Output (id, m) to ${\mathcal{A}_S}^H_{c(r(id))}$. Output $(id, |m|)$ to all adversaries ${\mathcal{A}_S}_i^H$.
3. If $s(id) \notin \mathcal{C}$, output (id, m) to $P_{r(id)}$. If $s(id) \in \mathcal{C}$, ${\mathcal{A}_S}^L$ may choose to output $(\mathsf{cheater}, s(id))$ to all parties instead.

• **Broadcast:** On input $(\mathsf{broadcast}, id, m)$ from $P_{s(id)}$ and $(\mathsf{broadcast}, id)$ from all (honest) parties:

1. For $s(id) \in \mathcal{C}$, let m be chosen by ${\mathcal{A}_S}^L$.
2. Output (id, m) to all adversaries ${\mathcal{A}_S}_i^H$.
3. If $s(id) \notin \mathcal{C}$, output (id, m) to all parties. If $s(id) \in \mathcal{C}$, $\mathcal{A}_S$ may choose to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).

• **Forward received message:** On input $(\mathsf{forward}, id)$ from $P_{r(id)}$ and on input $(\mathsf{forward}, id)$ from all (honest) parties, after (id, m) has been delivered to $P_{r(id)}$:

1. For $s(id), r(id) \in \mathcal{C}$, a new value for m is chosen by ${\mathcal{A}_S}^L$.
2. Output (id, m) to ${\mathcal{A}_S}^H_{c(f(id))}$. Output $(id, |m|)$ to all adversaries ${\mathcal{A}_S}_i^H$.
3. If $r(id) \notin \mathcal{C}$, output (id, m) to $P_{f(id)}$. If $r(id) \in \mathcal{C}$, ${\mathcal{A}_S}^L$ may choose to output $(\mathsf{cheater}, s(id))$ to all parties instead of (id, m).

• **Reveal received message:** On input (reveal, id) from all (honest) parties, such that $P_{f(id)}$ at any point received (id, m), output (id, m) to each party, and also to all adversaries ${\mathcal{A}_S}_i^H$.
If $s(id), r(id), f(id) \in \mathcal{C}$, then m is chosen by ${\mathcal{A}_S}^L$.
${\mathcal{A}_S}^L$ may output $(\mathsf{cheater}, k)$ to all parties for any $k \in \mathcal{C} \cap \{s(id), r(id), f(id)\}$. If $(\mathsf{cheater}, k)$ is output for all $k \in \{s(id), r(id), f(id)\}$, then no (id, m) is output to the parties.

Figure 5.9: Ideal functionality $\mathcal{F}_{transmit}{}^*$

In $\Pi_{transmit}^{cheap\ *}$, each party works with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$, and a party $f(id) \in [n]$ to whom the message should be forwarded by the receiver. Each party reacts to the same inputs as the parties of $\Pi_{transmit}$ given in Figure 4.12.

• **Initialization:** On input $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$, where $\mathsf{Dom}(\hat{s}) = \mathsf{Dom}(\hat{r}) = \mathsf{Dom}(\hat{f})$ assign the mappings $s \leftarrow \hat{s}$, $r \leftarrow \hat{r}$, $f \leftarrow \hat{f}$. The parties exchange their public keys that will be used to verify signatures later.

• **Transmit, forward:** On inputs $(\mathsf{transmit}, id, m)$, $(\mathsf{transmit}, id)$, $(\mathsf{forward}, id, m)$, $(\mathsf{forward}, id)$:

- *Cheap mode:* all parties act in exactly the same way as in $\Pi_{transmit}$.
- *Expensive mode:* all parties output $(id, \perp)$.

• **Broadcast and revealing:** On inputs $(\mathsf{broadcast}, id, m)$, $(\mathsf{broadcast}, id)$, (reveal, id), all parties act in exactly the same way as in $\Pi_{transmit}$.

Figure 5.10: The protocol $\Pi_{transmit}^{cheap\ *}$

Proposition 5.2. Let t be the upper bound on active coalition size. Assuming $t < n/2$, the protocol $\Pi_{transmit}^{cheap\ *}$ t-WCP-realizes $\mathcal{F}_{transmit}^{cheap\ *}$.

Proof. We use the simulator $\mathcal{S}_i = \mathcal{S}_{transmit}^{cheap\ *}(i)$ described in Figure 5.11-5.12. The simulator runs a local copy of $\Pi_{transmit}^{*}$.

Simulatability. The messages and signatures (m^*, σ_m^*) coming from the parties of $\mathcal{C}$ are obtained by $\mathcal{S}_i$ from $\mathcal{A}^L$. $\mathcal{S}_i$ delivers m^* to $\mathcal{F}_{transmit}^{*}$, so that the same m^* would be used in the ideal world. At any time when $m^* = \top$ is chosen, then $\mathcal{S}_i$ feeds $\top$ to $\mathcal{F}_{transmit}^{*}$ to get back the m that $\mathcal{A}_{S_i^H}$ was supposed to get. It then uses m in the simulation. By definition, $\mathcal{F}_{transmit}^{*}$ allows to decide later whether (id, m) or $(\mathsf{cheater}, s(id))$ should be output to the parties.

The simulator $\mathcal{S}_i$ gets from $\mathcal{F}_{transmit}^{*}$ all the messages received by P_k for $k \in \mathcal{C}_i$. For $k \notin \mathcal{C}_i$, only the message length is needed to simulate point-to-point channels, and this value is also provided by $\mathcal{F}_{transmit}^{*}$.

At any time when $\mathcal{S}_i$ detects the misbehaviour of a corrupted party P_k, it should send $(\mathsf{cheater}, k)$ to $\mathcal{F}_{transmit}^{*}$. The misbehaviour is detected as follows:

- *Transmission, forwarding:* $\mathcal{A}^L$ chooses $m^* = \perp$ or an invalid signature and message pair. There are no other restrictions on m. If $\mathcal{A}^L$ chooses $\top$, then there is definitely no misbehaviour.

- *Broadcast:* The same cases as for transmission. Additionally, $\mathcal{A}^L$ may choose a properly signed $m_k \neq m_{k'}$ for some $k, k' \notin \mathcal{C}$. If $\mathcal{A}^L$ chooses $\top$ for some of these values, $\mathcal{S}_i$ first sends them to $\mathcal{F}_{transmit}^{*}$ to get back m. Hence $\mathcal{S}_i$ is able to make all $m_k \neq m_{k'}$ comparisons.

- *Revealing:* The same cases as for the broadcast, and additionally $m_k \neq m$ for some $k \notin \mathcal{C}$, where m is the message that was actually transmitted. In this case, m comes from $\mathcal{F}_{transmit}^{*}$, so $\mathcal{S}_i$ is able to make all comparisons.

• **Initialization:** $\mathcal{S}_i$ gets (init, s, r, f) from ${\mathcal{F}_{transmit}}^*$. It simulates the parties exchanging their public keys. $\mathcal{A}^L$ provides public and secret keys for the parties $k \in \mathcal{C}$. $\mathcal{A}_i^H$ gets the secret keys of all parties $k \in \mathcal{C}_i$, and the public keys of all parties.

• **Secure transmission:**

1. Let $s(id) \notin \mathcal{C}$. If $r(id) \in \mathcal{C}_i$, $\mathcal{S}_i$ gets (id, m) from ${\mathcal{F}_{transmit}}^*$. It computes a signature σ_s on m and delivers (id, m, σ_s) to $\mathcal{A}_i^H$. If $r(id) \notin \mathcal{C}_i$, $\mathcal{S}_i$ gets $(id, |m|)$ from ${\mathcal{F}_{transmit}}^*$. It uses $|m|$ to model the view of $\mathcal{A}_i^H$ on messages moving through secure point-to-point channels between parties not in $\mathcal{C}_i$.
2. Let $s(id) \in \mathcal{C}$. $\mathcal{S}_i$ receives (m^*, σ_s^*) from $\mathcal{A}^L$. If $r(id) \notin \mathcal{C}$ and σ_s^* is not a valid signature of m^*, or $r(id) \in \mathcal{C}$ and $\mathcal{A}^L$ decided that it should complain, then $\mathcal{S}_i$ simulates $P_{r(id)}$ broadcasting (bad, id), and goes to expensive mode. Otherwise, $\mathcal{S}_i$ delivers (id, m^*) to ${\mathcal{F}_{transmit}}^*$. If $r(id) \in \mathcal{C}_i$, it delivers (id, m^*, σ_s^*) to $\mathcal{A}_i^H$. If $r(id) \notin \mathcal{C}_i$, it uses $|m^*|$ to model the view of $\mathcal{A}_i^H$ on messages moving through secure point-to-point channels.

At any time when $\mathcal{A}^L$ decides to choose $(m^*, \sigma_m^*) = \top$ for $s(id) \in \mathcal{C}$, then $\mathcal{S}_i$ delivers $(id, \top)$ to ${\mathcal{F}_{transmit}}^*$ as the input of $P_{s(id)}$, and it acts as in the case $s(id) \notin \mathcal{C}$. For simplicity, we will not mention modeling of point-to-point channels in the next points, although they are always present there.

• **Forwarding:** Simulated analogously to secure transmission.

• **Broadcast:**

1. If $s(id) \notin \mathcal{C}, \mathcal{S}_i$ gets (id, m) from ${\mathcal{F}_{transmit}}^*$. It takes $m_k = m$ for all $k \in [n]$, and generates a signature σ_{sk} on m_k. For $k \in \mathcal{C}_i$, it outputs all these (id, m_k, σ_{sk}) and $(id, m_k^*, \sigma_{sk}^*)$ to $\mathcal{A}_i^H$.
2. Let $s(id) \in \mathcal{C}$. $\mathcal{S}_i$ receives (m_k^*, σ_{sk}^*) from $\mathcal{A}^L$. If $k \notin \mathcal{C}$ and σ_{sk}^* is not a valid signature of $m_k'^*$, or $k \in \mathcal{C}$ and $\mathcal{A}^L$ decided that it should complain, then $\mathcal{S}_i$ simulates P_k broadcasting (bad, id). If at least t such broadcasts take place, then $\mathcal{S}_L^H$ delivers $(\mathcal{C}, p(id))$ to ${\mathcal{F}_{transmit}}^*$. If $k \in \mathcal{C}_i$, $\mathcal{S}_i$ delivers $(id, m_k'^*, \sigma_{sk}^*)$ to $\mathcal{A}_i^H$.
3. After that, in both cases, if $(\mathcal{C}, p(id))$ has not been output so far, $\mathcal{S}_i$ simulates all P_k for $k \in [n]$ sending to each other party the message (id, m_k, σ_{sk}) that it just received. Again, the pairs (m_k^*, σ_{sk}^*) for $k \in \mathcal{C}$ are chosen by $\mathcal{A}^L$. If any party P_k for $k \notin \mathcal{C}$ should have received (id, m, σ_s) and (id, m', σ_s') for $m \neq m'$, it simulates sending $(id, m, m', \sigma_s, \sigma_s')$ to each other party, and outputs $(\mathsf{cheater}, s(id))$ to ${\mathcal{F}_{transmit}}^*$. For $s(id) \in \mathcal{C}$, this happens if $m_k \neq m_{k'}$ is provided by $\mathcal{A}^L$ initially for some $k, k' \notin \mathcal{C}$. For $s(id) \notin \mathcal{C}$, $\mathcal{A}^L$ needs to get valid signatures of $P_{s(id)}$ on m' to have such a situation, and it happens only with negligible probability.
4. If no $(\mathsf{cheater}, s(id))$ has been output to ${\mathcal{F}_{transmit}}^*$, then all m_k^* received by all honest parties are equal to the same value m'. For $s(id) \notin \mathcal{C}$, it should be $m' = m$. For $s(id) \in \mathcal{C}$, ${\mathcal{F}_{transmit}}^*$ is waiting for m. $\mathcal{S}_i$ outputs to ${\mathcal{F}_{transmit}}^*$ the message (id, m'), where m' is the value accepted by all honest parties in the real protocol.

$\mathcal{A}^L$ may choose $(m_k^*, \sigma_{mk}^*) = \top$ for $s(id) \in \mathcal{C}$ for some k. In this case $\mathcal{S}_i$ delivers $\top$ to ${\mathcal{F}_{transmit}}^*$ and gets back m, substituting $\top$ with m in the simulation. If only some of the messages were $\top$, it may happen that $m_k^* \neq m$. In this case, $\mathcal{S}_i$ delivers $(\mathsf{cheater}, s(id))$ to ${\mathcal{F}_{transmit}}^*$, so that $(\mathsf{cheater}, s(id))$ is output to all parties instead of (id, m).

Figure 5.11: The simulator ${\mathcal{S}_{transmit}^{cheap}}^*(i)$ (initialization, transmit, forward, broadcast)

• **Reveal received message:**

1. If at least one of $s(id), r(id), f(id) \notin \mathcal{C}$, then $\mathcal{S}_i$ gets (id, m) from ${\mathcal{F}_{transmit}}^*$, where m is the value that was actually transmitted. If all $s(id), r(id), f(id) \in \mathcal{C}$, then ${\mathcal{F}_{transmit}}^*$ waits for m^* from the adversary.
2. The simulation proceeds similarly to the broadcast. In the real protocol, we now observe the situation where $(id, m, \sigma_s, \sigma_r, \sigma_f)$ and $(id, m', \sigma'_s, \sigma'_r, \sigma'_f)$ are received by some honest party, where all the signatures are valid, but $m \neq m'$. We want it to be detectable by $\mathcal{S}_i$. Similarly to common broadcast, unless $s(id), r(id), f(id) \in \mathcal{C}$, $\mathcal{A}^L$ should be able to come up with an alternative set of signatures σ'_s, σ'_r, σ'_f, and it may happen only with negligible probability. If all of them are corrupt, then message revealing is equivalent to broadcasting by a corrupted sender.

If the revealing of m by $f(id)$ fails, then it is repeated by $r(id)$ and $s(id)$. The simulations are analogous to revealing by $f(id)$.

Figure 5.12: The simulator ${\mathcal{S}^{cheap}_{transmit}}^*(i)$ (revealing received messages)

Correctness. Similarly to $\Pi_{transmit}$ of UC model, honest majority assumption ensures that the broadcasts and revealings either succeed, or all honest parties agree that the sender is corrupted. In the latter case, $\mathcal{S}_i$ delivers $(\mathsf{cheater}, s(id))$ to ${\mathcal{F}_{transmit}}^*$. For transmissions and forwarding, the message delivery is not guaranteed, but if it fails, then all honest parties output $(id, \perp)$. In this case, $\mathcal{S}_i$ delivers $(id, \perp)$ to ${\mathcal{F}_{transmit}}^*$. □

Committing randomness. First of all, we define a functionality ${\mathcal{F}_{rnd}}^*$ that we use to generate committed randomness. It will be used many times in our protocol, and properly shared randomness seems to be very important against attacks specific to WCP. Compared to $\mathcal{F}_{rnd}$ of Figure 4.23, we allow ${\mathcal{F}_{rnd}}^*$ internally compute linear combinations and truncations of committed values, and also to open them, which makes it more similar to $\mathcal{F}_{commit}$ of Figure 4.16. The protocol steps implementing these additional functions are analogous to Π_{commit} of Figure 4.17, and the only difference is that they are based on ${\mathcal{F}^{cheap}_{transmit}}^*$ instead of $\mathcal{F}_{transmit}$. However, the definition of opening in the ideal functionality is different, officially allowing to reveal the leaves of derivation trees of commitments to *honest* parties. The same has actually happened in $\mathcal{F}_{commit}$, but UC model did not capture this leakage, and hence it was not defined in $\mathcal{F}_{commit}$. We will take this leakage into account when using ${\mathcal{F}_{rnd}}^*$ as a subroutine in our protocols.

The ideal functionality ${\mathcal{F}_{rnd}}^*$ is depicted in Figure 5.13. Since ${\mathcal{F}^{cheap}_{transmit}}^*$ works only in cheap mode, we allow that randomness generation may fail, so we are going to use it only during preprocessing.

The protocol ${\Pi_{rnd}}^*$ implementing ${\mathcal{F}_{rnd}}^*$ is given in Figure 5.14. Differently from Π_{rnd} that was based on $\mathcal{F}_{commit}$, we have inlined the implementations of commit and priv_open of Π_{commit} directly into Π_{rnd} of Figure 4.24.

The functionality ${\mathcal{F}_{rnd}}^*$ works with unique identifiers id, encoding the party $p(id)$ to which the committed randomness will be known, and the bit length $m(id)$ of the randomness. It stores an array $comm$ of already generated and committed randomness, as well as their linear combinations and truncations. For each id, it stores the derivation tree $deriv[id]$ showing how $comm[id]$ was computed from the other committed values.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$, assign the mappings $m \leftarrow \hat{m}$, $p \leftarrow \hat{p}$. Deliver $(\mathsf{init}, \hat{m}, \hat{p})$ to all adversaries ${\mathcal{A}_S}_i^H$.

• **Extension:** On input $(\mathsf{ext}, \hat{m}, \hat{p})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$ and $\mathsf{Dom}(\hat{m}) \cap \mathsf{Dom}(m) = \emptyset$, extend the mappings $m \leftarrow m \cup \hat{m}$, $p \leftarrow p \cup \hat{p}$ over the new domain $\mathsf{Dom}(\hat{m}) \cup \mathsf{Dom}(m)$. Deliver $(\mathsf{ext}, \hat{m}, \hat{p})$ to all adversaries ${\mathcal{A}_S}_i^H$.

• **Randomness commitment:** On input (rnd, id) from all (honest) parties, generate a random bitstring r of length $m(id)$. Output r to ${\mathcal{A}_S}_{c(p(id))}^H$. Wait whether ${\mathcal{A}_S}^L$ inputs $(id, \top)$ or (stop, id). On input (stop, id) from ${\mathcal{A}_S}^L$, stop the functionality and output $(id, \bot)$ to all parties. Otherwise, assign $comm[id] \leftarrow r$, and output r to $P_{p(id)}$. Output $(\mathsf{confirmed}, id)$ to all parties.

• **Compute Linear Combination and Truncation:** On inputs $(\mathsf{lc}, \vec{c}, \vec{id}, id)$ and $(\mathsf{trunc}, m', id, id')$ from all (honest) parties, compute the linear combination and the truncation respectively, similarly to $\mathcal{F}_{commit}$ of Figure 4.16, updating the arrays $comm[id']$ and $deriv[id']$.

• **Weak Open:** On input $(\mathsf{weak\_open}, id)$ from all (honest) parties, output $comm[id]$ to all ${\mathcal{A}_S}_i^H$. If ${\mathcal{A}_S}^L$ sends (stop, id), then output $(id, \bot)$ to each party. Otherwise, output $(id, comm[id])$ to each party.

• **Open:** On input (open, id) from all (honest) parties, output $comm[id]$, to all ${\mathcal{A}_S}_i^H$. Wait whether ${\mathcal{A}}^L$ inputs $\top$ or some messages $(\mathsf{cheater}, k)$ for $k \in \mathcal{C}$. Unless it inputs $(\mathsf{cheater}, p(id))$, output $(id, comm[id])$ to all parties.

For each $(\mathsf{cheater}, k)$ input by ${\mathcal{A}}^L$, output $(\mathsf{cheater}, k)$ to all parties. If $(\mathsf{cheater}, k)$ is output for *all* $k \in \mathcal{C}$, taking into account the messages $(\mathsf{cheater}, k)$ output during the previous openings, output all the leaves of $deriv[id]$ to all ${\mathcal{A}_S}_i^H$ for $i \neq 1$.

• **Privately Open:** On input $(\mathsf{priv\_open}, id, id')$ from all (honest) parties, if $deriv[id] = id$, output $(id, id', comm[id])$ to ${\mathcal{A}_S}_{c(p(id'))}^H$. Wait whether ${\mathcal{A}}^L$ inputs $\top$ or (stop, id). If (stop, id) comes, output $(id, id', \bot)$ to each party. Otherwise, write $comm[id'] = comm[id]$, output $(id, id', comm[id])$ to $P_{p(id')}$, and output $(\mathsf{confirmed}, id)$ to all parties.

• **Stopping:** At any time when ${\mathcal{A}_S}^L$ delivers (stop, id) [resp. (stop, id, id')], output $(id, \bot)$ [resp. $(id, id', \bot)$] to each party.

Figure 5.13: Ideal functionality ${\mathcal{F}_{rnd}}^*$

In $\Pi_{rnd}{}^*$, each party works locally with unique identifiers id, encoding the bit size $m(id)$ of the ring in which the value is shared, and the party $p(id)$ committed to the value. The parties use a linear (n,t)-threshold sharing scheme with $t = \lceil n/2 \rceil + 1$. Each party P_k stores its own local copy of an array $comm$ into which it writes its shares. Each party stores a term $deriv[id]$ (represented by a tree whose leaves are the initial commitments, and the inner nodes are the lc, trunc operations applied to them) to remember in which way each $comm[id]$ has been computed. For the initially committed values, let $deriv[id] = id$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$, each party assigns the mappings $m \leftarrow \hat{m}$, $p \leftarrow \hat{p}$. For all $id \in \mathsf{Dom}(m) = \mathsf{Dom}(p)$, it defines mappings s, r, and f, such that $s(id^{jk}) \leftarrow j$, $r(id^{jk}) \leftarrow k$, $f(id^{jk}) \leftarrow p(id)$, and $s(id_j^k) \leftarrow p(id)$, $r(id_j^k) \leftarrow k$, $f(id_j^k) \leftarrow j$ for all $id \in \mathsf{Dom}(m) = \mathsf{Dom}(p)$, $j,k \in [n]$. In addition, it defines the senders $s(id_k^{\mathsf{bc}}) \leftarrow k$ for the broadcasts (used for share opening). It sends (init, s, r, f) to $\mathcal{F}_{transmit}^{cheap}{}^*$.

• **Extension:** On input $(\mathsf{ext}, \hat{m}, \hat{p})$, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$ and $\mathsf{Dom}(\hat{m}) \cap \mathsf{Dom}(m) = \emptyset$, extend the mappings $m \leftarrow m \cup \hat{m}$, $p \leftarrow p \cup \hat{p}$. For the new identifiers id, initialize a new instance of $\mathcal{F}_{transmit}^{cheap}{}^*$, similarly to the initialization.

Randomness commitment: On input (rnd, id), if $comm[id]$ has not been defined yet:

1. Each party P_j, $j \neq p(id)$ (actually, $t+1$ parties are sufficient), generates a random value $r_j \in \mathbb{Z}_{2^{m(id)}}$, shares it as $(r_j^k)_{k\in[n]} = \mathsf{classify}(r_j)$, writes $comm[id] \leftarrow (r_j^k)_{k\in[n]}$, and sends $(\mathsf{transmit}, id^{jk}, r_j^k)$ to $\mathcal{F}_{transmit}^{cheap}{}^*$ for all $k \in [n]$.
2. Upon receiving (id^{jk}, r_j^k) from $\mathcal{F}_{transmit}^{cheap}{}^*$ for all $j \in [n]\backslash\{p(id)\}$, P_k sends $(\mathsf{forward}, id^{jk})$ to $\mathcal{F}_{transmit}^{cheap}{}^*$. It writes $comm[id^{jk}] \leftarrow r_j^k$.
3. Upon receiving (id^{jk}, r_j^k) from $\mathcal{F}_{transmit}^{cheap}{}^*$ for all $k \in [n], j \in [n] \setminus \{p(id)\}$, $P_{p(id)}$ checks if the shares r_j^k are consistent. If they are, $P_{p(id)}$ computes $r^k = \sum_{j\in[n]\setminus\{p(id)\}} r_j^k$, and sends $(\mathsf{transmit}, id_i^k, r^k)$ for all $i \in [n] \setminus \{p(id)\}$ to $\mathcal{F}_{transmit}$ (only one message is actually transmitted for all $i \in [n]$, it is just recorded under n different identifiers). If they are not, $P_{p(id)}$ broadcasts (bad, id).
4. Upon receiving (id_i^k, r'^k) from $\mathcal{F}_{transmit}$, P_k checks if $r^k = r'^k$ for $r^k = \sum_{j\in[n]\setminus\{p(id)\}} r_j^k$. If it is, it assigns $comm[id^k] \leftarrow r^k$. If $r^k \neq r'^k$, it broadcasts (bad, id). Upon receiving (bad, id), each party outputs $(id, \perp)$.

• **Compute Linear Combination and Truncation:** On inputs $(\mathsf{lc}, \vec{c}, \vec{id}, id)$ and $(\mathsf{trunc}, m', id, id')$, the parties act in exactly the same way as in Π_{commit} of Figure 4.17, computing these two operations locally on shares, without any interaction.

• **Open, Weak Open, Privately Open:** On inputs (open, id), $(\mathsf{weak\_open}, id)$, and $(\mathsf{priv\_open}, id)$, the parties act in exactly the same way as in Π_{commit} of Figure 4.17, now using $\mathcal{F}_{transmit}^{cheap}{}^*$ instead of $\mathcal{F}_{transmit}$.

• **Stopping:** At any time when the transmission of $\mathcal{F}_{transmit}^{cheap}{}^*$ fails (due to missing expensive mode), or (bad, id) is broadcast, each party outputs $(id, \perp)$.

Figure 5.14: The protocol $\Pi_{rnd}{}^*$

• **Initialization and extension:** $\mathcal{S}_i$ gets (init, m, p) or (ext, m, p) from ${\mathcal{F}_{rnd}}^*$. It simulates initialization of ${\mathcal{F}_{transmit}^{cheap}}^*$.

• **Randomness Commitment:** On input (rnd, id):

1. First of all, $\mathcal{S}_i$ needs to simulate r_j^k for $k \in \mathcal{C}_i$. For $j \notin \mathcal{C}$, $\mathcal{S}_i$ samples $r_j^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$. For $j \in \mathcal{C}$, $\mathcal{S}_i$ gets r_j^k from $\mathcal{A}^L$. If $\mathcal{A}^L$ chooses $\top$ instead of some r_j^k, then $\mathcal{S}_i$ generates $r_j^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$ itself. As the result, $\mathcal{A}_i^H$ receives all n shares for $j \in \mathcal{C}_i$, but only up to $t-1$ shares for $j \notin \mathcal{C}_i$.
2. In the real protocol, all the shares r_j^k should now be forwarded to $P_{p(id)}$. If $p(id) \in \mathcal{C}_i$ then $\mathcal{S}_i$ needs to simulate the forwarding to $\mathcal{A}_i^H$. It gets r from ${\mathcal{F}_{rnd}}^*$. $\mathcal{S}_i$ chooses the remaining shares r_j^k for $k \notin \mathcal{C}_i$ in such a way that $\sum_{j\in[n]} \mathsf{declassify}(r_j^k)_{k\in[n]} = r$. This is done similarly to $\mathcal{F}_{rnd}$ of Figure 4.23.
3. After $\mathcal{S}_i$ gets all r_j^k for $j \in \mathcal{C}$, it simulates the consistently check by $P_{p(id)}$, sending (stop) to ${\mathcal{F}_{rnd}}^*$ if it does not pass. If the check passes, $\mathcal{S}_i$ simulates transmitting r_j^k back to $k \in \mathcal{C}_i$. If $p(id) \in \mathcal{C}$, then $\mathcal{A}^L$ may decide to send (stop) to ${\mathcal{F}_{rnd}}^*$.
4. After all the transmissions and forwardings have been simulated, $\mathcal{S}_i$ checks if all the parties P_k for $k \notin \mathcal{C}$ should have received $r_j^k = r_j'^k$. If the check fails, $\mathcal{S}_i$ sends stop to ${\mathcal{F}_{rnd}}^*$ and simulates broadcasting a complaint using ${\mathcal{F}_{transmit}^{cheap}}^*$.

• **Compute Linear Combination and Truncation:** Similarly to $\mathcal{S}_{commit}$ of Figure 4.19, $\mathcal{S}_i$ computes these operations locally on the shares of $\mathcal{C}_i$.

• **Stopping:** At any time when ${\mathcal{F}_{transmit}^{cheap}}^*$ outputs $(id, \perp)$ or $(\mathsf{cheater}, k)$, $\mathcal{S}_i$ sends (stop) to ${\mathcal{F}_{rnd}}^*$.

Figure 5.15: The simulator ${\mathcal{S}_{rnd}}^*(i)$ (init, stop, commit, local operations)

Proposition 5.3. Let t be the upper bound on active coalition size. Assuming $t < n/2$, the protocol ${\Pi_{rnd}}^*$ t-WCP-realizes ${\mathcal{F}_{rnd}}^*$ in ${\mathcal{F}_{transmit}^{cheap}}^*$-hybrid model.

Proof. We use the simulator $\mathcal{S}_i = {\mathcal{S}_{rnd}}^*(i)$ described in Figure 5.15. The simulator runs a local copy of ${\Pi_{rnd}}^*$.

Simulatability. During the randomness generation, $\mathcal{S}_i$ should be able to simulate the shares r_j^k for $k, j \in \mathcal{C}_i$. Since $\mathcal{A}_i^H$ gets at most t shares of each value r_j, only those r_j for which $j \in \mathcal{C}_i$ may be seen by $\mathcal{A}_i^H$ in their entirety. In addition, there are up to $t-1$ values r_j whose shares $\mathcal{A}^L$ generates for $j \in \mathcal{C}$, all of these may be already known to $\mathcal{Z}$ (if $\mathcal{A}^L$ chooses $\top$ instead of some r_j^k, then $\mathcal{S}_i$ generates r_j^k from the appropriate distribution itself).

Since at least $t+1$ parties contribute r_j, there is at least one $j \notin \mathcal{C}_i \cup \mathcal{C}$. From this r_j, at most $t-1$ shares can be known to $\mathcal{Z}$. Hence as far as r is not opened, $\mathcal{S}_i$ may generate all the shares r_j^k from uniform distribution, and $\mathcal{Z}$ still does not have enough shares r_j^k to infer anything about r. As soon as r is opened, there is at least one share r_j^k not known to $\mathcal{Z}$ yet, that can now be adjusted in such a way that $\mathsf{declassify}(r_j^k)_{k\in\mathcal{H}} = r - \sum_{j\in[n]\setminus\{p(id)\}} r_j$. This can be done similarly to the proof of Lemma 4.4, generating the missing shares from uniform distribution until only one share remains, that is uniquely determined by r.

• **Weak Open:** $\mathcal{S}_i$ gets (id, x) from ${\mathcal{F}_{rnd}}^*$. $\mathcal{S}_i$ needs to simulate broadcasting x^k by P_k. If $p(id) \in \mathcal{C}_i$, then $\mathcal{S}_i$ already knows x and all x^k, so it may use them in the simulation. Otherwise, $\mathcal{S}_i$ should already have simulated at most $t-1$ shares x^k for $k \in \mathcal{C}_i$. If it has less than $t-1$ shares, it generates shares y^k of leaves of $deriv[id]$ from uniform distribution, computes x^k from them, and then adjusts the remaining shares in such a way that $\mathsf{declassify}(x^k)_{k\in[n]} = x$ (they are now uniquely determined). $\mathcal{S}_i$ does it for all parties, including the corrupted ones. Now all shares x^k have been generated by $\mathcal{S}_i$.

$\mathcal{A}^L$ chooses x^{*k} that should be broadcast for $k \in \mathcal{C}$. $\mathcal{S}_i$ simulates the broadcast of x^{*k} for $k \in \mathcal{C}$, and x^k for $k \notin \mathcal{C}$. $\mathcal{S}_i$ checks if $x^{*k} = x^k$. If $x^{*k} \neq x^k$, $\mathcal{S}_i$ outputs (stop) to ${\mathcal{F}_{rnd}}^*$, and simulates the complaint of P_k.

Open: First of all, $\mathcal{S}_i$ gets (id, x) from ${\mathcal{F}_{rnd}}^*$.

- $\mathcal{S}_i$ simulates sending $(\mathsf{reveal}, id^k_{jk})$ for all $j, k \in [n]$ to ${\mathcal{F}^{cheap}_{transmit}}^*$. For $p(id) \notin \mathcal{C}$, all the shares of parties not in $\mathcal{C}_i$ should be generated by $\mathcal{S}_i$ itself. These shares are simulated in the same way as in the case of weak opening.

  It is more complex with the leaves of $deriv[id]$. Since at most $t-1$ leaf shares belonging to $\mathcal{C}$ may be revealed, the simulation is easy for $\mathcal{C}_i = \mathcal{C}$, but for $\mathcal{C}_i \neq \mathcal{C}$, $\mathcal{S}_i$ should have already simulated one share of each leaf to $\mathcal{A}^H_i$. Now it may need to output all t shares of the leaves of $deriv[id]$ to $\mathcal{A}^H_i$. In this case, $\mathcal{S}_i$ receives the leaves of $deriv[id]$ from ${\mathcal{F}_{rnd}}^*$.
- If the opening succeeds for all r^k_j, in the real protocol each party reconstructs $r = \sum_j \mathsf{declassify}(r^k_j)$. For $j \in \mathcal{C}$, the shares may be inconsistent. This event is immediately noticed by $\mathcal{S}_i$ since the set of inconsistent shares has been entirely chosen by $\mathcal{A}^L$, so $\mathcal{S}_i$ outputs $(\mathsf{cheater}, p(id))$ to $\mathcal{F}_{rnd}$.

• **Privately Open:**

- If $p(id) \in \mathcal{C}$, then $\mathcal{S}_i$ gets (id, x) from ${\mathcal{F}_{rnd}}^*$. The shares x^k such that $\mathsf{declassify}(x^k)_{k\in[n]} = x$ are constructed similarly to the previous two openings.
- If $p(id) \notin \mathcal{C}_i$, then $\mathcal{S}_i$ does not get (id, x) from ${\mathcal{F}_{rnd}}^*$. However, it still needs to simulate the behaviour of $k \in \mathcal{C}$. Since there are at most $t-1$ such parties P_k, it is sufficient that $\mathcal{S}_i$ generates the first $t-1$ shares similarly to the previous two openings.

After that, $\mathcal{S}_i$ only needs to simulate transmissions and forwardings of these shares using $\mathcal{F}_{transmit}$. After all the transmissions and forwardings have been simulated, $\mathcal{S}_i$ checks if all the parties P_k for $k \notin \mathcal{C}$ should have received $x^k_j = x'^k$. If the check fails, $\mathcal{S}_i$ sends stop to ${\mathcal{F}_{rnd}}^*$ and simulates broadcasting a complaint using ${\mathcal{F}^{cheap}_{transmit}}^*$.

Figure 5.16: The simulator ${\mathcal{S}_{rnd}}^*(i)$ (openings)

As far as r is not opened, since $r^k = \sum_{j\in[n]\setminus\{p(id)\}} r^k_j$, there is at least one r^k_j generated by an honest party, serving as a mask for r^k. As the result, the shares r^k that are given to $k \notin \mathcal{C}_i$ come from the same distribution as if they were all generated by an honest party (i.e. any set of $t-1$ of them is distributed uniformly, and all other shares are uniquely determined by these $t-1$ uniform shares). If $p(id) \notin \mathcal{C}_i$, then $\mathcal{A}^H_i$ does not know anything neither about the randomness r nor its shares of $k \notin \mathcal{C}_i$, regardless of r^k_j chosen by $\mathcal{A}^L$, even if $p(id) \in \mathcal{C}$. In the simulation of weak opening, it will be very important that the shares of parties $k \notin \mathcal{C}_i$ look uniformly distributed to $\mathcal{A}^H_i$, unless it gets at least t shares.

During the weak opening, ${\mathcal{F}_{rnd}}^*$ first outputs (id, x) to $\mathcal{S}_i$. It needs to simulate opening of x^k by P_k. If $p(id) \in \mathcal{C}_i$, then $\mathcal{S}_i$ has already known x and all x^k before, and all of them have already been simulated to $\mathcal{A}_i^H$, so $\mathcal{S}_i$ may use them again. Otherwise, $\mathcal{S}_i$ should already have simulated at most $t-1$ shares x^k for $k \in \mathcal{C}_i$. If $|\mathcal{C}_i| = t-1$, then knowing x uniquely determines all the other shares. If $|\mathcal{C}_i| < t-1$, then $\mathcal{S}_i$ needs to generate some of the shares itself. We show that generating $y^k \xleftarrow{\$} \mathbb{Z}_{2^{m(id')}}$ for leaves id' of $deriv[id]$ and computing $x^k = deriv[id](y_1, \ldots, y_l)$ is what $\mathcal{Z}$ expects to see in the real protocol.

- Let x be the initially generated randomness. During the generation of x, $\mathcal{A}^L$ has no control over the shares x^k. Hence unless $p(id) \in \mathcal{C}_i$, all the shares of $k \notin \mathcal{C}_i$ are distributed uniformly in the initial ring.

- All leaves y^k of $deriv[id]$ correspond to some initially generated randomness, which is distributed uniformly. The linear combinations and truncations are uniquely determined by y^k.

All the shares for private opening are generated similarly to the weak opening. Their transmissions and forwardings using ${\mathcal{F}_{transmit}^{cheap}}^*$ are analogous to the transmissions of r^k during the randomness commitment.

In strong opening, the shares of x^k can be simulated for x in a similar way. Additionally, if $i \neq 1$, $\mathcal{S}_i$ may need to reveal up to t shares y^k of leaves y of $deriv[id]$ to $\mathcal{A}_j^H$. The value y is provided by ${\mathcal{F}_{rnd}}^*$ that outputs y to all $\mathcal{S}_j^H$ for $j \neq 1$, so that the missing t-th share can be computed directly from y and the other $t-1$ shares y^k.

$\mathcal{S}_i$ should always be able to detect the misbehaviour of $\mathcal{A}^L$ without seeing any information that the other simulators $\mathcal{S}_j^H$ received from ${\mathcal{F}_{rnd}}^*$. It may happen in the following cases:

- The message $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{transmit}$.

- During the randomness generation, the sender P_j has transmitted inconsistent shares. For $j \notin \mathcal{C}$, this never happens. For $j \in \mathcal{C}$, all shares r_j^k are given to $\mathcal{S}_i$ by $\mathcal{A}^L$, or are generated by $\mathcal{S}_i$ itself (for $r_j^{*k} = \top$) so the check is easy to do.

- During the weak opening, $\mathcal{S}_i$ first generates x^k itself from appropriate distribution, as discussed above. Hence $x^k = x^{*k}$ is a valid check with respect to what $\mathcal{Z}$ awaits from the real protocol execution. It may seem a bit strange that fresh x^k may be generated already *after* $\mathcal{A}^L$ has chosen x^{*k}. In this case, since x^k is generated inside the protocol ${\Pi_{rnd}}^*$, unless $x^{*k} = \top$, $\mathcal{A}^L$ expects to get $x^{*k} = x^k$ with the same probability as if x^k was generated independently from x^{*k}.

Table 5.1: Calls of ${\mathcal{F}_{transmit}^{cheap}}^*$ for different functionalities of ${\Pi_{rnd}}^*$ with N-bit values

input	called ${\mathcal{F}_{transmit}^{cheap}}^*$ functionalities
rnd	$\mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot N}^{\otimes n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes n(t+1)}$
weak_open	$\mathsf{bc}_{\mathsf{sh}_n \cdot N}^{\otimes n}$
open	$\mathsf{rev}_{\mathsf{sh}_n \cdot N}^{\otimes n}$
priv_open	$\mathsf{fwd}_{\mathsf{sh}_n \cdot N}^{\otimes n} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot N}^{\otimes n}$
pcommit, lc, trunc	–

Correctness. ${\mathcal{F}_{rnd}}^*$ outputs r to $P_{p(id)}$. $\mathcal{S}_i$ generates r_j^k of $j \notin \mathcal{C}$ in such a way that $r = \sum_j \mathsf{declassify}(r_j^k)_{k \in \mathcal{H}}$, so this value is the same in both worlds. If a linear combination or a truncation is computed, then $x = \mathsf{declassify}(x_j^k)_{k \in \mathcal{H}}$ still holds for the new values x, due to linearity of secret sharing scheme (the proof is analogous to Lemma 4.2).

Since $\mathcal{S}_i$ sends $(\mathsf{cheater}, k)$ and $(id, \perp)$ to ${\mathcal{F}_{transmit}^{cheap}}^*$ in all cases where it simulates outputting $(\mathsf{cheater}, k)$ and $(id, \perp)$ in the real protocol, these outputs are also consistent. □

Observation 5.1. Compared to corresponding UC protocol Π_{rnd} of Figure 4.24, we have in fact not modified the randomness generation protocol, and we have built its WCP version on top of ${\mathcal{F}_{transmit}^{cheap}}^*$ implemented by ${\Pi_{transmit}}^*$ that we have not modified at all. Therefore, the cost of randomness generation is the same as in Table 4.3. The only difference is that at least $t + 1$ values r_j are now necessary instead of t. The costs of computing linear combinations and the openings are exactly the same as in Table 4.2, since we have not modified the corresponding protocols at all. The cost summary is given in Table 5.1.

Expensive Mode of ${\mathcal{F}_{transmit}}^*$. We are now ready to define a protocol ${\Pi_{transmit}}^*$ implementing ${\mathcal{F}_{transmit}}^*$ in WCP model. It is given in Figure 5.17. The main idea behind the protocol is that each pair P_i, P_j of parties holds mutual randomness $q_{ij}(id)$ for each message identifier id such that P_i is the sender and P_j is the receiver. This randomness should not be known to any other party. It is generated in the preprocessing phase, and it is committed using (n, t)-threshold sharing, so that authenticity of q_{ij} can be proven later. For this purpose, the parties use ${\mathcal{F}_{rnd}}^*$. During the execution of ${\mathcal{F}_{transmit}}^*$, if the transmission of m fails, then the sender is required to broadcast $m' = m + q_{ij}$ to all parties. No party P_k for $k \neq j$ is able to read this message since q_{ij} serves as a mask. If the message m that was transmitted in this way needs to be revealed, then the parties open q_{ij} using ${\mathcal{F}_{rnd}}^*$, and each party computes $m = m' - q_{ij}$ for the previously broadcast m'.

In $\Pi_{transmit}{}^*$, each party works locally with unique message identifiers id, encoding a sender $s(id) \in [n]$, a receiver $r(id) \in [n]$, and a party $f(id) \in [n]$ to whom the message should be forwarded by the receiver. It reacts to the same party inputs as $\Pi_{transmit}$ of Figure 4.12, and we only present here those calls for which the behaviour of $\Pi_{transmit}{}^*$ is different from $\Pi_{transmit}$. The protocol uses $\mathcal{F}_{rnd}{}^*$ as a subroutine.

• **Initialization:** On input $(\mathsf{init}, \hat{s}, \hat{r}, \hat{f})$, where $\mathsf{Dom}(\hat{s}) = \mathsf{Dom}(\hat{r}) = \mathsf{Dom}(\hat{f})$ assign the mappings $s \leftarrow \hat{s}$, $r \leftarrow \hat{r}$, $f \leftarrow \hat{f}$. For each $id \in \mathsf{Dom}(\hat{s})$, generate the identifiers id_r, id'_r and id_f, id'_f (these will be used for message transmission and forwarding respectively). Assign $p(id_r) \leftarrow s(id)$, $p(id'_r) = p(id_f) \leftarrow r(id)$, $p(id'_f) \leftarrow f(id)$, and $m(id_r) = m(id'_r) = m(id_f) = m(id'_f) \leftarrow m$, where m is the expected message length. Send $(\mathsf{rnd}, id_r, id'_r)$ and $(\mathsf{rnd}, id_f, id'_f)$ to $\mathcal{F}_{rnd}{}^*$. Now for any message that P_i is going to transmit to P_j, there is committed randomness q_{ij} known only to P_i and P_j.

• **Secure transmit:** On inputs $(\mathsf{transmit}, id, m)$, $(\mathsf{transmit}, id)$:

- *Cheap mode:* All parties act in exactly the same way as in $\Pi_{transmit}$ given in Figure 4.12.
- *Expensive mode:* Used if $P_{r(id)}$ complains about $P_{s(id)}$ not following the protocol.

  1. On input $(\mathsf{transmit}, id, m)$ the party $P_{s(id)}$ takes the next value $q = q_{s(id),r(id)}$ that it obtained from $\mathcal{F}_{rnd}{}^*$, computes $m' = m + q$ over $\mathbb{Z}_{2^{|m|}}$, signs (id, m') to obtain signature σ_s. It sends (id, m', σ_s) to each other party.
  2. Each party P_i sends (id, m', σ_s) to $P_{r(id)}$. If P_i does not receive (id, m', σ_s), it sends a signature γ_i on $(\mathsf{cheater}, s(id))$ to all parties. If an honest party receives at least t such messages, it outputs $(\mathsf{cheater}, s(id))$ to $\mathcal{Z}$.
  3. On input $(\mathsf{transmit}, id)$, $P_{r(id)}$ expects a message (id, m', σ_s) from each P_i, where σ_s is a valid signature of $P_{s(id)}$ on (id, m'). If it arrives from some P_i, then $P_{r(id)}$ reads the next $q = q_{s(id),r(id)}$ it obtained from $\mathcal{F}_{rnd}{}^*$, and outputs $(id, m' - q)$.

• **Forwarding:** analogous to secure transmit.

• **Broadcast:** On inputs $(\mathsf{broadcast}, id, m)$, $(\mathsf{broadcast}, id)$, all parties act in exactly the same way as in $\Pi_{transmit}$ given in Figure 4.12.

• **Reveal received message:** On input (reveal, id):

- If $m(id)$ was sent using cheap mode of $\Pi_{transmit}{}^*$, act in the same way as in $\Pi_{transmit}$.
- If $m(id)$ was sent using expensive mode of $\Pi_{transmit}{}^*$, act in the same way as in $\Pi_{transmit}$ to open the value $m' = m + q$. Send $(\mathsf{open}, (id, s(id), r(id)))$ to $\mathcal{F}_{rnd}{}^*$, getting $q = q_{s(id),r(id)}$. Output $(id, m' - q)$ to $\mathcal{Z}$.

Figure 5.17: The protocol $\Pi_{transmit}{}^*$

• **Initialization:** $\mathcal{S}_i$ gets (init, s, r, f) from ${\mathcal{F}_{transmit}}^*$. It simulates initialization and execution of ${\mathcal{F}_{rnd}}^*$ to share the values q_{kj} among P_k and P_j. All the values q_{kj}, where either $k \in \mathcal{C}_i$ or $j \in \mathcal{C}_i$, are chosen by $\mathcal{S}_i$. The other values q_{kj} do not need to be simulated yet, and they will be used up later as random masks.

• **Secure transmit:**

1. *Cheap mode:* $\mathcal{S}_i$ behaves in the same way as ${\mathcal{S}^{cheap}_{transmit}}^*$ of Figure 5.11.
2. *Expensive mode:*

   (a) Let $s(id) \notin \mathcal{C}$. If $r(id) \in \mathcal{C}_i$, $\mathcal{S}_i$ gets (id, m) from ${\mathcal{F}_{transmit}}^*$. It computes $m' \leftarrow m + q(id)$. If $r(id) \notin \mathcal{C}_i$, $\mathcal{S}_i$ gets $(id, |m|)$ from ${\mathcal{F}_{transmit}}^*$. It generates $m' \xleftarrow{\$} \mathbb{Z}_{|m|}$. In both cases, it generates a signature σ_s on m', and delivers (id, m', σ_s) to $\mathcal{A}_i^H$.

   (b) Let $s(id) \in \mathcal{C}$. $\mathcal{S}_i$ receives (m'^*_k, σ^*_{sk}) from $\mathcal{A}^L$. If $k \notin \mathcal{C}$ and σ^*_{sk} is not a valid signature of m'^*_k, or $k \in \mathcal{C}$ and $\mathcal{A}^L$ decided that it should complain, then $\mathcal{S}_i$ simulates P_k broadcasting (bad, id). If at least t such broadcasts take place, then $\mathcal{S}_i$ delivers $(\mathsf{cheater}, p(id))$ to ${\mathcal{F}_{transmit}}^*$. Otherwise, $\mathcal{S}_i$ takes $m'^* = m'^*_k$ of some $k \notin \mathcal{C}$ that has not complained (the one that $P_{r(id)}$ would choose in the real protocol).

   If $r(id) \in \mathcal{C}_i$ or $s(id) \in \mathcal{C}_i$, then $q(id)$ has already been generated by $\mathcal{S}_i$. If $r(id) \notin \mathcal{C}_i$, then $\mathcal{S}_i$ generates $q(id) \xleftarrow{\$} \mathbb{Z}_{|m'^*|}$. It computes $m^* = m'^* - q(id)$, and sends m^* to ${\mathcal{F}_{transmit}}^*$. If $k \in \mathcal{C}_i$, $\mathcal{S}_i$ delivers $(id, m'^*_k, \sigma^*_{sk})$ to $\mathcal{A}_i^H$.

   At any time when $\mathcal{A}^L$ decides to choose $(m^*, \sigma^*_m) = \top$ for $s(id) \in \mathcal{C}$, then $\mathcal{S}_i$ delivers $(id, \top)$ to ${\mathcal{F}_{transmit}}^*$ as the input of $P_{s(id)}$, and it acts as in the case $s(id) \notin \mathcal{C}$. In the expensive mode, it is done only after a certain (m'^*, σ^*_s) has been chosen for $P_{r(id)}$ (it may be that $(m'^*_k, \sigma^*_{sk}) = \top$ only for some k).

• **Broadcast:** $\mathcal{S}_i$ behaves in the same way as ${\mathcal{S}^{cheap}_{transmit}}^*$.

• **Reveal received message:** $\mathcal{S}_i$ behaves in the same way as ${\mathcal{S}^{cheap}_{transmit}}^*$, but now it should take into account that a message might have been transmitted or forwarded in the expensive mode.
If m was sent using expensive mode of ${\Pi_{transmit}}^*$, opening of q should be simulated instead. In this case, $\mathcal{S}_i$ gets the revealed message m from ${\mathcal{F}_{transmit}}^*$ regardless of corruptions. At the same time, $m' = m + q$ should already have been simulated to $\mathcal{A}_i^H$ during the corresponding transmission. $\mathcal{S}_i$ simulates opening the value q using ${\mathcal{F}_{rnd}}^*$, which is not under control of corrupted parties, so $\mathcal{S}_i$ may simulate the opening in such a way that $q = m' - m$.

Figure 5.18: The simulator ${\mathcal{S}_{transmit}}^*(i)$

Proposition 5.4. Let t be the upper bound on active coalition size. If $t < n/2$, then the protocol ${\Pi_{transmit}}^*$ t-WCP-realizes ${\mathcal{F}_{transmit}}^*$ in ${\mathcal{F}_{rnd}}^*$-hybrid model.

Proof. We use the simulator $\mathcal{S}_i = {\mathcal{S}_{transmit}}^*(i)$ described in Figure 5.18. The simulator runs a local copy of ${\Pi_{transmit}}^*$.

Simulatability. In the beginning, $\mathcal{S}_i$ simulates ${\mathcal{F}_{transmit}}^*$ generating mutual randomness q_{kj} for the parties P_k and P_j. For $k \in \mathcal{C}_i$ and $j \in \mathcal{C}_i$, the values q_{kj} should be revealed to $\mathcal{A}_i^H$. However, the other q_{kj} do not need to be simulated yet.

The cheap mode is simulated in the same way as by ${\mathcal{S}^{cheap}_{transmit}}^*$. In the expensive mode, $\mathcal{S}_i$ needs to simulate $m' = m + q$. If $s(id) \in \mathcal{C}_i$ or $r(id) \in \mathcal{C}_i$, then $\mathcal{S}_i$

obtains m from ${\mathcal{F}_{transmit}}^*$ (for $r(id) \in \mathcal{C}_i$), or uses the view of $P_{s(id)}$ to compute m itself (for $s(id) \in \mathcal{C}_i$) so there are no problems with computing $m' = m + q$, where q has already been simulated to $\mathcal{A}_i^H$ before. If $s(id), r(id) \notin \mathcal{C}_i$, then $\mathcal{S}_i$ does not get m from ${\mathcal{F}_{transmit}}^*$. For $s(id) \notin \mathcal{C}$ there is no hope for $\mathcal{S}_i$ to get m at all. However, in this case, q has not been simulated to $\mathcal{A}_i^H$ yet, so $\mathcal{S}_i$ may sample m' from uniform distribution. Since the mask q comes from uniform distribution according to the properties of ${\mathcal{F}_{rnd}}^*$, and each such q is used only once, the actual value $m + q$ would also come from the uniform distribution, since the addition takes place in the finite ring $\mathbb{Z}_{2^{|m|}}$.

In the expensive mode, choosing any $m' \neq \perp$ is acceptable. Hence it is easy for $\mathcal{S}_i$ to detect whether $\mathcal{A}^L$ is cheating, just by checking if $m'^* = \perp$.

Correctness. The delivery of messages in the cheap mode works correctly by Proposition 5.2. In addition, we need to show that delivery of $m' = m + q$ instead of m keeps the correctness. Although the assisting parties cannot verify if m that is masked by q is properly formatted, we do not require this check since $\mathcal{F}_{transmit}$ by definition allows m to be chosen by $\mathcal{A}^L$ for $s(id) \in \mathcal{C}$ (the format should be defined separately for ${\mathcal{F}_{transmit}}^*$ if needed). Hence m' can be arbitrary without violating correctness assumptions.

Since $\mathcal{S}_i$ sends $(\mathsf{cheater}, k)$ and $(id, \perp)$ to ${\mathcal{F}_{transmit}^{cheap}}^*$ in all cases where it simulates outputting $(\mathsf{cheater}, k)$ and $(id, \perp)$ in the real protocol, these outputs are also consistent. □

Observation 5.2. Compared to corresponding UC protocol $\Pi_{transmit}$, we have modified the expensive mode in such a way that the costs of expensive transmissions are now the same as the costs of broadcasts. This results in expensive transmission to be n times more costly than it was in UC model. Otherwise, we may still use the results of Table 4.1 to estimate the costs of ${\Pi_{transmit}}^*$.

The additional overhead is that $\mathcal{F}_{transmit}$ now has a preprocessing phase, in which mutual randomness should be generated. For each pair of parties that are going to communicate M bits in the online phase, M bits of such randomness should be generated. According to Table 5.1, its cost is $\mathsf{tr}_{\mathsf{sh}_n \cdot M}^{\otimes n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M}^{\otimes 2n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M}^{\otimes 2n(t+1)}$, where the multiplier 2 comes from two private openings to two parties.

5.4.3 Covert Adversaries

We modify the protocol Π_{verify} of Figure 4.31, getting a new protocol ${\Pi_{verify}}^*$ that t-WCP implements ${\mathcal{F}_{verify}}^*$. For this, we first construct t-WCP secure implementations for the building block functionalities ${\mathcal{F}_{commit}^{weak}}^*$ and ${\mathcal{F}_{pre}}^*$. For a covertly secure protocol, it is sufficient that only the optimistic mode will not leak any data to honest parties. This makes the protocols much simpler compared to the active adversary case.

We define ${\mathcal{F}_{commit}^{weak}}^*$ similarly to the UC functionality $\mathcal{F}_{commit}$ given in Figure 4.16, but with the following changes:

1. We discard the functionality priv_open. In protocols of Chapter 4, the only use of priv_open was inside Π_{rnd} implementing $\mathcal{F}_{rnd}$. We have already constructed ${\Pi_{rnd}}^*$ independently from ${\mathcal{F}_{commit}^{weak}}^*$, and there will be no other use for priv_open in our protocols.

2. We add the randomness commitment functionality rnd to ${\mathcal{F}_{commit}^{weak}}^*$. Since ${\Pi_{rnd}}^*$ is not built on top of ${\mathcal{F}_{commit}^{weak}}^*$, it will be easier for the WCP analogue ${\Pi_{verify}}^*$ of Π_{verify} to commit randomness using ${\mathcal{F}_{commit}^{weak}}^*$ directly, and not use $\mathcal{F}_{rnd}$, as Π_{verify} of Figure 4.31 did.

3. On input (open, id), the functionality explicitly leaks some data to the honest parties. If we use the UC protocol Π_{commit} of Figure 4.17 without modification, we cannot avoid this leakage, but as we show further, it will be sufficient to implement the verification for a *covert* adversary. If the adversary is *active*, we will need to modify Π_{commit}, so that it would implement open as it is defined in $\mathcal{F}_{commit}$, without additional leakage to the honest parties.

The ideal functionality ${\mathcal{F}_{commit}^{weak}}^*$ is given in Figure 5.19. On input (open, id), ${\mathcal{F}_{commit}^{weak}}^*$ allows to leak some additional information to honest parties. However, this leakage is accompanied by a public accusation of the party that caused the leakage. Since a covert adversary never attempts to cheat if it will be caught, this information will never be leaked in the real protocol.

Our new version of ${\Pi_{commit}^{weak}}^*$ implementing ${\mathcal{F}_{commit}^{weak}}^*$ ensures than the shares issued to the parties (and whose authenticity can be proven later) are distributed uniformly. For this, the parties use preshared randomness that is generated in the preprocessing phase by ${\mathcal{F}_{rnd}}^*$. If a party wants to commit to a value x, it broadcasts $x' = x + q$, where q is properly shared randomness. For mutual commitments, the randomness q should be known to both parties $P_{p(id)}$ and $P_{p(id')}$ that get committed to the same value. Using preshared randomness allows to avoid WCP-specific attacks based on bad sharing that may eventually leak information to some honest party. The protocol ${\Pi_{commit}^{weak}}^*$, built on top of ${\mathcal{F}_{transmit}}^*$ and ${\mathcal{F}_{rnd}}^*$, is given in Figure 5.20-5.21.

Proposition 5.5. Let t be the upper bound on active coalition size. If $t < n/2$, the protocol ${\Pi_{commit}^{weak}}^*$ t-WCP-realizes ${\mathcal{F}_{commit}^{weak}}^*$ in ${\mathcal{F}_{transmit}}^*$-${\mathcal{F}_{rnd}}^*$-hybrid model.

Proof. We use the simulator $\mathcal{S}_i = {\mathcal{S}_{commit}^{weak}}^*(i)$ described in Figure 5.22. The simulator runs a local copy of ${\Pi_{commit}^{weak}}^*$ and ${\mathcal{F}_{transmit}}^*$.

$\mathcal{F}_{commit}^{weak}{}^{*}$ works with unique identifiers id, encoding the bit size $m(id)$ of the ring $\mathbb{Z}_{2^{m(id)}}$ in which the value is committed, the party $p(id)$ committed to the value, and the randomness $r(id)$ that will be generated already during the initialization, but will be actually committed on inputs (rnd, id) later. The commitments are stored in an array $comm$, and their derivations in an array $deriv$. For each id, the term $deriv[id]$ is a tree whose leaves are the initial commitments, and the inner nodes are the lc, trunc operations applied to them. For the initially committed values, $deriv[id] = id$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$, define the mappings $m \leftarrow \hat{m}, p \leftarrow \hat{p}$. For all identifiers id, generate $r(id) \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$. Deliver $(\mathsf{init}, \hat{m}, \hat{p})$ to all adversaries ${\mathcal{A}_S}_i^H$. For all id, deliver $r(id)$ to $\mathcal{A}_{c(p(id))}^H$.

• **Extension:** On input $(\mathsf{ext}, \hat{m}, \hat{p})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$ and $\mathsf{Dom}(\hat{m}) \cap \mathsf{Dom}(m) = \emptyset$, extend the mappings $m \leftarrow m \cup \hat{m}$, $p \leftarrow p \cup \hat{p}$ over the new domain $\mathsf{Dom}(\hat{m}) \cup \mathsf{Dom}(m)$. Deliver $(\mathsf{ext}, \hat{m}, \hat{p})$ to all adversaries ${\mathcal{A}_S}_i^H$.

• **Public Commit:** On input $(\mathsf{pcommit}, id, x)$ from all (honest) parties, if $x \in \mathbb{Z}_{2^{m(id)}}$, write $comm[id] \leftarrow x$, and output $(\mathsf{confirmed}, id)$ to all parties and all adversaries ${\mathcal{A}_S}_i^H$. Output x to all ${\mathcal{A}_S}_i^H$.

• **Commit:** On input (commit, id, x) from $P_{p(id)}$ and (commit, id) from all (honest) parties, if $x \in \mathbb{Z}_{2^{m(id)}}$, write $comm[id] \leftarrow x$, and output $(\mathsf{confirmed}, id)$ to all parties and all ${\mathcal{A}_S}_i^H$. If $p(id) \in \mathcal{C}$, then x is chosen by ${\mathcal{A}_S}^L$, who may alternatively deliver to $\mathcal{F}_{commit}^{weak}{}^{*}$ a message $(\mathsf{cheater}, p(id))$ that is output by $\mathcal{F}_{commit}^{weak}{}^{*}$ to all parties.

• **Mutual Commit:** On input $(\mathsf{mcommit}, id, id', x)$ from $P_{p(id)}$, $(\mathsf{mcommit}, id, id', x')$ from $P_{p(id')}$, and $(\mathsf{mcommit}, id, id')$ from all (honest) parties, output (id, id', x, x') to ${\mathcal{A}_S}_{c(p(id))}^H$. Compare x and x'. If $x = x'$, then write $comm[id], comm[id'] \leftarrow x$, and output $(\mathsf{confirmed}, id, id')$ to all parties and all adversaries ${\mathcal{A}_S}_i^H$. If $x \neq x'$, output $(id, id', \perp)$ to all parties and to all ${\mathcal{A}_S}_i^H$.

If $p(id) \in \mathcal{C}$ [resp. $p(id') \in \mathcal{C}$], then ${\mathcal{A}_S}^L$ chooses x [resp. x']. Alternatively, it may deliver to $\mathcal{F}_{commit}{}^{*}$ a message $(\mathsf{cheater}, p(id))$ or $(\mathsf{cheater}, p(id'))$ that is output by $\mathcal{F}_{commit}{}^{*}$ to all parties.

• **Random Commit:** On input (rnd, id) from all (honest) parties, assign $comm[id] \leftarrow r(id)$, and output $r(id)$ to $P_{p(id)}$. Output $(\mathsf{confirmed}, id)$ to all parties and to all ${\mathcal{A}_S}_i^H$.

• **Compute Linear Combination: and Truncation:** Since no outputs for the adversary are produced, the definitions are exactly the same as for $\mathcal{F}_{commit}$ of Figure 4.16.

• **Weak Open:** On input $(\mathsf{weak\_open}, id)$ from all (honest) parties, output $comm[id]$ to all ${\mathcal{A}_S}_i^H$. If ${\mathcal{A}_S}^L$ sends (stop), then output $(id, \perp)$ to each party. Otherwise, output $(id, comm[id])$ to each party.

• **Open:** On input (open, id) from all (honest) parties, after $(\mathsf{weak\_open}, id)$ has failed, output $comm[id]$, to all ${\mathcal{A}_S}_i^H$. Wait until $\mathcal{A}^L$ inputs either $\top$ or some messages $(\mathsf{cheater}, k)$ for $k \in \mathcal{C}$. Unless it inputs $(\mathsf{cheater}, p(id))$, output $(id, comm[id])$ to all parties. For each $(\mathsf{cheater}, k)$ input by $\mathcal{A}^L$, output $(\mathsf{cheater}, k)$ to all parties. If $(\mathsf{cheater}, k)$ is output for *all* $k \in \mathcal{C}$, output all the leaves of $deriv[id]$ to all ${\mathcal{A}_S}_i^H$ for $i \neq 1$.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ is output to all parties, do not accept any inputs including id s.t. $p(id) = k$ anymore. Let $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$, $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$.

Figure 5.19: Ideal functionality $\mathcal{F}_{commit}^{weak}{}^{*}$

In $\Pi_{commit}^{weak}{}^{*}$, each party works locally with unique identifiers id, encoding the bit size $m(id)$ of the ring in which the value is shared, and the party $p(id)$ committed to the value. Each party P_k stores an array $pubv$, into which it writes certain published values that will be later used for opening the commitments. For the new identifiers id that will store the new values computed from the committed values, store a term $deriv[id]$ (represented by a tree whose leaves are commitments, and the inner nodes are operations applied to them) to remember in which way they were computed. For the committed values, let $deriv[id] = id$. $\Pi_{commit}^{weak}{}^{*}$ works on top of $\mathcal{F}_{transmit}{}^{*}$ and $\mathcal{F}_{rnd}{}^{*}$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p})$, define a mapping s, such that $s(id) \leftarrow p(id)$, for all $id \in \mathsf{Dom}(p)$ (all these identifiers will be used only for broadcasts). Send $(\mathsf{init}, s, \perp, \perp)$ to $\mathcal{F}_{transmit}{}^{*}$ (the mappings r and f are not defined at all).
Send $(\mathsf{init}, \hat{m}, \hat{p})$ to $\mathcal{F}_{rnd}{}^{*}$. For all id, send (rnd, id) to $\mathcal{F}_{rnd}{}^{*}$, generating the committed randomness that is known only to $p(id)$. For all (id, id') pairs, send (rnd, id) followed by $(\mathsf{priv\_open}, id, id')$ to $\mathcal{F}_{rnd}{}^{*}$. As the result, there is a committed randomness $q(id, id')$ known only to $p(id)$ and $p(id')$.

• **Extension:** On input $(\mathsf{ext}, \hat{m}, \hat{p})$, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p})$ and $\mathsf{Dom}(\hat{m}) \cap \mathsf{Dom}(m) = \emptyset$, extend the mappings $m \leftarrow m \cup \hat{m}$, $p \leftarrow p \cup \hat{p}$. Initialize a new instance of $\mathcal{F}_{transmit}{}^{*}$ similarly to the initialization. Send $(\mathsf{ext}, \hat{m}, \hat{p})$ followed by $(\mathsf{priv\_open}, id, id')$ to $\mathcal{F}_{rnd}{}^{*}$, resulting in committed randomness $q(id, id')$ for all new identifier pairs id, id'.

• **Public Commit:** On input $(\mathsf{pcommit}, id, x)$, each party P_k writes $comm[id] \leftarrow x^k$, where x^k is computed directly from x according to any preagreed sharing.

• **Commit:**

1. On input (commit, id, x), $P_{p(id)}$ computes $x' \leftarrow x + q(id)$. It sends $(\mathsf{broadcast}, id, x')$ to $\mathcal{F}_{transmit}$.
2. On input (commit, id), P_k waits for (id, x') from $\mathcal{F}_{transmit}$.

If the broadcast succeeds, each party writes $pubv[id] \leftarrow x'$, $deriv[id] \leftarrow id$.

• **Random Commit:** On input (rnd, id), each party writes $pubv[id] \leftarrow 0$, $deriv[id] \leftarrow id$.

• **Mutual Commit:**

1. On input $(\mathsf{mcommit}, id, id', x)$, $P_{p(id)}$ computes $\hat{x} \leftarrow x + q(id, id')$. It sends $(\mathsf{broadcast}, (id, id'), \hat{x})$ to $\mathcal{F}_{transmit}$.
2. On input $(\mathsf{mcommit}, id, id', x')$, $P_{p(id')}$ waits for $(id, \hat{x})$ from $\mathcal{F}_{transmit}$. If $\hat{x} \neq x' + q(id, id')$, it broadcasts (bad, id, id').
3. On input $(\mathsf{mcommit}, id, id')$, P_k waits for $(id, \hat{x})$ from $\mathcal{F}_{transmit}$.

If the broadcast of $\hat{x}$ succeeds, and (bad, id, id') is not broadcast, then each party writes $pubv[id] \leftarrow x'$, $deriv[id] \leftarrow id$. Otherwise, it outputs $(id, id', \perp)$.

• **Compute Linear Combination:** On input $(\mathsf{lc}, \vec{c}, \vec{id}, id')$, where $\ell := |\vec{c}| = |\vec{id}|$, and $p' = p(id_i)$ are the same for all $i \in \{1, \ldots, \ell\}$, for $m' \leftarrow \min(\{m(id_i) \mid i \in \{1, \ldots, \ell\}\})$, each party P_k

1. sends $(\mathsf{lc}, \vec{c}, \vec{id}, id')$ to $\mathcal{F}_{rnd}{}^{*}$;
2. writes $pubv[id'] \leftarrow (\sum_{i=1}^{\ell} c_i \cdot pubv[id_i]) \bmod 2^m$;
3. assigns $m(id') \leftarrow m'$, $p(id') \leftarrow p'$, $deriv[id'] \leftarrow \mathsf{lc}(\vec{c}, \vec{id})$.

• **Compute Truncation:** On input $(\mathsf{trunc}, m', id, id')$, where $m(id) \geq m' \in \mathbb{N}$, each party P_k

1. sends $(\mathsf{trunc}, m', id, id')$ to $\mathcal{F}_{rnd}{}^{*}$;
2. writes $pubv[id'] \leftarrow pubv[id] \bmod 2^{m'}$;
3. assigns $m(id') \leftarrow m'$, $p(id') \leftarrow p(id)$, $deriv[id'] \leftarrow \mathsf{trunc}(m', id)$.

Figure 5.20: Real Protocol $\Pi_{commit}^{weak}{}^{*}$ (init, commit, local operations)

• **Weak Open:** On input $(\mathsf{weak\_open}, id)$:

1. Each party sends $(\mathsf{weak\_open}, id)$ to ${\mathcal{F}_{rnd}}^*$.
2. Upon receiving (id, q) from ${\mathcal{F}_{rnd}}^*$, each party computes $x = pubv[id] - q$, and outputs x to $\mathcal{Z}$. If ${\mathcal{F}_{rnd}}^*$ returns $(id, \perp)$, each party outputs $(id, \perp)$ to $\mathcal{Z}$.

• **Open:** On input (open, id):

1. Each party sends (open, id) to ${\mathcal{F}_{rnd}}^*$.
2. Upon receiving (id, q) from ${\mathcal{F}_{rnd}}^*$, each party computes $x \leftarrow pubv[id] - q$, and outputs x to $\mathcal{Z}$.

• **Cheater detection:** At any time when a party receives $(\mathsf{cheater}, k)$ from ${\mathcal{F}_{transmit}}^*$, it outputs $(\mathsf{cheater}, k)$ to $\mathcal{Z}$. After outputting $(\mathsf{cheater}, k)$ to $\mathcal{Z}$, it does not accept any inputs including id s.t. $p(id) = k$ anymore, and treats P_k as if it has left the protocol, i.e. $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 5.21: Real Protocol ${\Pi_{commit}^{weak}}^*$ (openings)

Simulatability. In the beginning, while generating the randomness using ${\mathcal{F}_{rnd}}^*$, $\mathcal{S}_i$ only needs to simulate the randomness that is output to $P_{p(id)}$ for $p(id) \in \mathcal{C}_i$ (for mutual randomness, also $p(id') \in \mathcal{C}_i$). Similarly to ${\mathcal{S}_{transmit}}^*(i)$, $\mathcal{S}_i$ does not have to simulate the randomness of the other parties yet. It will be used as a mask for their future commitments.

During the commitments, $\mathcal{S}_i$ gets from ${\mathcal{F}_{commit}^{weak}}^*$ only the values x for $p(id) \in \mathcal{C}_i$, or $p(id') \in \mathcal{C}_i$. If both $p(id), p(id') \notin \mathcal{C}_i$, then $\mathcal{S}_i$ needs to simulate the broadcast of x' without knowing x. Since $x' = x + q(id)$, and $q(id)$ is being used first time as a mask which is not known to $\mathcal{A}_i^H$, it is sufficient to sample x' uniformly.

- For $p(id) \notin \mathcal{C}$, $\mathcal{S}_i$ will need to decide on the precise value for $q(id)$ later, when opening will be required. Besides opening, $q(id)$ is not used anywhere else.

- For $p(id) \in \mathcal{C}$, the value $x^{*k} \neq \top$ provided by $\mathcal{A}^L$ immediately requires $\mathcal{S}_i$ to determine the value of $q(id)$, so that $x = x^{*k} - q(id)$ could be committed to ${\mathcal{F}_{commit}^{weak}}^*$. We need to show that this x is the value that $\mathcal{A}^L$ expects to be committed. For $i = 1$, $\mathcal{A}^L$ should already have known $q(id)$ before, so it expects that the committed value will be $x = x'^* - q(id)$. For $i \neq 1$, since $\mathcal{A}^L$ has no idea about the randomness $q(id)$ for $p(id) \notin \mathcal{C}_i$, and x'^* would have been generated independently from $q(id)$ in the real protocol, the distribution of x would be the same.

For all openings, revealing $q(id)$ is fully reduced to ${\mathcal{F}_{rnd}}^*$. It is now the right time for $\mathcal{S}_i$ to choose $q(id)$ for $p(id), p(id') \notin \mathcal{C}_i$. First of all, $\mathcal{S}_i$ receives x from ${\mathcal{F}_{commit}^{weak}}^*$. It has already computed $pubv[id]$ as a linear combination on the leaves of $deriv[id]$, so it takes $q(id) = pubv[id] - x$. We show that this $q(id)$ is what $\mathcal{Z}$ expects to see.

Let $comm[id]$, $pubv$, $deriv$ the local arrays of $\mathcal{S}_i$.

• **Initialization and extension:** $\mathcal{S}_i$ gets (init, m, p) and (ext, m, p) from ${\mathcal{F}_{commit}^{weak}}^*$. It initializes ${\mathcal{F}_{transmit}}^*$, and initializes/extends ${\mathcal{F}_{rnd}}^*$, generating the randomness $q = q(id, id')$ for $p(id) \in \mathcal{C}_i$ and $p(id') \in \mathcal{C}_i$. If $p(id), p(id') \notin \mathcal{C}_i$, then q may be chosen later.

• **Public commit:** On input $(\mathsf{pcommit}, id, x)$, $\mathcal{S}_i$ gets x from $\mathcal{F}_{commit}$. It computes $(x^k)_{k\in[n]} = \mathsf{classify}(x)$ according to the preagreed sharing and writes $comm[id] \leftarrow x$.

• **Commit:** $\mathcal{S}_i$ gets x from ${\mathcal{F}_{commit}^{weak}}^*$ for $p(id) \in \mathcal{C}_i$, and computes $x' \leftarrow x + q(id)$, where $q(id)$ comes from ${\mathcal{F}_{rnd}}^*$. For $p(id) \notin \mathcal{C}_i$, it generates a random x' and simulates use of $\mathcal{F}_{transmit}$ broadcasting x'. $\mathcal{A}^L$ may provide x'^* for $p(id) \in \mathcal{C}$. If $\mathcal{A}^L$ provides $\top$, then $\mathcal{S}_i$ uses x' that it has generated before.
$\mathcal{S}_i$ now has to output something to ${\mathcal{F}_{commit}^{weak}}^*$ for $p(id) \in \mathcal{C}$. If $i = 1$, then $\mathcal{S}_i$ has already simulated $q(id)$ to $\mathcal{A}_i^H$ before, so the committed value should be $x = x'^* - q(id)$. If $i \neq 1$, then $\mathcal{S}_i$ delivers $\top$ to ${\mathcal{F}_{commit}^{weak}}^*$ if $x'^* = \top$, and if $x'^* \neq \top$, it generates a new randomness $q(id)$ and computes $x \leftarrow x'^* - q(id)$.

• **Random Commit:** There is no interaction between the parties. $\mathcal{S}_i$ assigns $pubv[id] \leftarrow 0$, $deriv[id] \leftarrow id$ locally.

• **Mutual Commit:** The simulation is similar to (commit, id, x). The difference is that $\mathcal{S}_i$ may need to simulate the broadcast of (bad, id, id') if $\hat{x} \neq x' + q(id, id')$ is broadcast. Although $\mathcal{S}_i$ does not know x', it gets $(id, id', \bot)$ from $\mathcal{F}_{commit}$ if $x \neq x'$.

• **Compute Linear Combination and Truncation:** $\mathcal{S}_i$ locally performs the computations for all $k \in \mathcal{C}_i$. No outputs are produced. New values are assigned to the local array $comm[id]$, and the derivation tree is stored in $deriv[id]$.

• **Weak Open:** The value x to be opened is given to $\mathcal{S}_i$ by ${\mathcal{F}_{commit}^{weak}}^*$. $\mathcal{S}_i$ simulates sending $(\mathsf{weak\_open}, id)$ to ${\mathcal{F}_{rnd}}^*$, making it open $q = pubv[id] - x$. If (stop) comes to ${\mathcal{F}_{rnd}}^*$, then $\mathcal{S}_i$ outputs (stop) to ${\mathcal{F}_{commit}^{weak}}^*$.

• **Open:** The value x to be opened is given to $\mathcal{S}_i$ by ${\mathcal{F}_{commit}^{weak}}^*$. $\mathcal{S}_i$ simulates sending (open, id) to ${\mathcal{F}_{rnd}}^*$, making it open $q = pubv[id] - x$. By definition, $\mathcal{A}^L$ is able to manipulate ${\mathcal{F}_{rnd}}^*$ in such a way that it opens the leaves of $deriv[id]$ to $\mathcal{A}_i^H$ for $i \neq 1$. $\mathcal{S}_i$ gets all these values from ${\mathcal{F}_{commit}^{weak}}^*$.

• **Cheater Detection:** At any time when $\mathcal{S}_i$ notices that $(\mathsf{cheater}, k)$ should be output to each honest party in ${\Pi_{commit}^{weak}}^*$, then it discards P_k from their local runs of ${\Pi_{commit}^{weak}}^*$, i.e. $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$ and $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$. $\mathcal{S}_i$ delivers $(\mathsf{cheater}, k)$ to ${\mathcal{F}_{commit}^{weak}}^*$.

Figure 5.22: The simulator ${\mathcal{S}_{commit}^{weak}}^*(i)$

- For the initial commitments x (i.e. $deriv[id] = id$), $pubv[id]$ is the initial value x' whose broadcast was simulated at some point before, where $\mathcal{Z}$ was expecting $x' = x + q$. Hence opening $q = x' - x$ is what it expects.

- For a more complex $deriv[id]$, we have $pubv[id] = deriv[id](x'_1, \ldots, x'_l)$ computed from values x'_i whose broadcasts were simulated to $\mathcal{Z}$ at some point before. In all cases, $\mathcal{Z}$ was expecting to see $x'_i = x_i + q_i$, where x_i is some leaf of $deriv[id]$ and q_i is the randomness that corresponds to it. By definition, ${\mathcal{F}_{commit}^{weak}}^*$ should have opened $x = deriv[id](x_1, \ldots, x_l)$. By linearity of $deriv[id]$, it should be $pubv[id] - x = deriv[id](x'_1 - x_1, \ldots, x'_l - x'_l) = deriv[id](q_1, \ldots, q_l)$. Hence $\mathcal{S}_i$ has opened $q =$

$deriv[id](q_1, \ldots, q_l)$. This is what the output of ${\mathcal{F}_{rnd}}^*$ is expected to be, since the parties in the real protocol have applied the operations of $deriv[id]$ to $q_1, \ldots, q_l$ using ${\mathcal{F}_{rnd}}^*$.

Strong opening may require the leaves of $deriv[id]$ to be output to adversaries $\mathcal{A}_i^H$ for $i \neq 1$. In this case, $\mathcal{S}_i$ gets these leaves from ${\mathcal{F}_{commit}^{weak}}^*$. Otherwise, it is similar to the weak opening.

Correctness. For each x that ${\mathcal{F}_{commit}^{weak}}^*$ outputs to P_k for $k \in \mathcal{C}_i$ as a commitment, $\mathcal{S}_i$ has simulated exactly the same x in its interaction with $\mathcal{A}_i^H$.

The randomness commitment is a particular case of ordinary commitment, where $x' = 0$. This choice results in $0 = x' = x + q$, where x is treated as the committed randomness. Since q is uniformly distributed randomness, so is $r = -q$. Formally, to ensure that ${\mathcal{F}_{commit}^{weak}}^*$ would output *exactly the same* randomness r, $\mathcal{S}_i$ should let ${\mathcal{F}_{rnd}}^*$ generate $q = -r$ during the initialization. Since generation of q pushed into the preprocessing, $\mathcal{S}_i$ uses $q = -r(id)$, where $r(id)$ is the value that it gets from ${\mathcal{F}_{commit}^{weak}}^*$. The same value $r(id)$ will be used by ${\mathcal{F}_{commit}^{weak}}^*$ as $comm[id]$ for the randomness commitment.

For the openings, $\mathcal{S}_i$ has caused ${\mathcal{F}_{rnd}}^*$ to output q such that $x + q = pubv[id]$, where $pubv[id]$ has been computed by $\mathcal{S}_i$ from $deriv[id]$ in the same way as all honest parties would compute it. In the real world, all (honest) parties would output the value $pubv[id] - q = x$, which is the same as ${\mathcal{F}_{commit}^{weak}}^*$ outputs. □

Observation 5.3. Differently from corresponding UC protocol, instead of transmitting n shares, the committing party makes a broadcast. This modifies the costs of commitments. From the definition of ${\Pi_{commit}^{weak}}^*$, we may read out the new complexities of ${\mathcal{F}_{commit}^{weak}}^*$ operations. They are given in Table 5.2. Similarly to Table 4.2, we use tr_M, bc_M, fwd_M, rev_M denote the calls of transmit, broadcast, forward, reveal respectively on an M-bit message, and sh_n the number of times the bit width the value shared among n parties is smaller than the bit width of its one share.

In addition, ${\Pi_{commit}^{weak}}^*$ now requires precomputation of randomness in the initialization phase. This is done by using cheap ${\mathcal{F}_{transmit}}^*$ in the preprocessing phase. Using $\mathcal{F}_{transmit}$ in cheap mode does not in turn require more preprocessing.

Comparing these values with Table 4.2, we see that the multiplicative overhead of the committing functionalities is n in the online phase, and it is $3n$ in the offline phase. The opening and the computing of linear combinations incur no overheads.

Public Randomness Generation Protocol ${\mathcal{F}_{pubrnd}}^*$

Similarly to Chapter 4, we will also need a functionality for generating public randomness. The ideal functionality ${\mathcal{F}_{pubrnd}}^*$ is given in Figure 5.23, and it does not have any modifications except the new adversary ports of WCP model.

Table 5.2: Calls of ${\mathcal{F}_{transmit}}^*$ for different functionalities of ${\Pi^{weak}_{commit}}^*$ with N-bit values

functionality	Online calls	Offline calls
commit	$\mathsf{bc}_{\mathsf{sh}_n \cdot N}$	$\mathsf{tr}^{\otimes n(t+1)}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{fwd}^{\otimes n(t+1)}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{tr}^{\otimes n(t+1)}_{\mathsf{sh}_n \cdot N}$
mcommit	$\mathsf{bc}_{\mathsf{sh}_n \cdot N}$	$\mathsf{tr}^{\otimes n(t+1)}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{fwd}^{\otimes 2n(t+1)}_{\mathsf{sh}_n \cdot N} \oplus \mathsf{tr}^{\otimes 2n(t+1)}_{\mathsf{sh}_n \cdot N}$
weak_open	$\mathsf{bc}^{\otimes n}_{\mathsf{sh}_n \cdot N}$	–
open	$\mathsf{rev}^{\otimes n}_{\mathsf{sh}_n \cdot N}$	–
lc, trunc	–	–

The functionality ${\mathcal{F}_{pubrnd}}^*$ works with unique identifiers id, encoding the bit length $m(id)$ of the randomness.

• **Initialization:** On input $(\mathsf{init}, \hat{m})$, assign the mapping $m \leftarrow \hat{m}$. Deliver $\hat{m}$ to all ${\mathcal{A}_S}^H_i$.

• **Randomness commitment:** On input (pubrnd, id) from all (honest) parties, generate a random value $r \in \mathbb{Z}_{2^{m(id)}}$. Output (id, r) to each party, and also to ${\mathcal{A}_S}^H_i$. Alternatively, if $\mathcal{C} \neq \emptyset$, ${\mathcal{A}_S}^L$ may choose to output $(\mathsf{cheater}, k)$ for $k \in \mathcal{C}$ to each party instead.

• **Cheater detection:** On input $(\mathsf{cheater}, k)$ from ${\mathcal{A}_S}^L$ for $k \in \mathcal{C}$, output $(\mathsf{cheater}, k)$ to all parties. Let $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}, \mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 5.23: Ideal functionality ${\mathcal{F}_{pubrnd}}^*$

The protocol ${\Pi_{pubrnd}}^*$ is given in Figure 5.24. Similarly to ${\Pi_{rnd}}^*$, we will use sharing directly, instead of building ${\Pi_{pubrnd}}^*$ on top of ${\mathcal{F}^{weak}_{commit}}^*$ or ${\mathcal{F}_{rnd}}^*$, since we want to exclude overheads.

Proposition 5.6. Let t be the upper bound on active coalition size. Assuming $t < n/2$, the protocol ${\Pi_{pubrnd}}^*$ t-WCP-realizes ${\mathcal{F}_{pubrnd}}^*$ in ${\mathcal{F}_{transmit}}^*$-hybrid model.

Proof. We use the simulator $\mathcal{S}_i = {\mathcal{S}_{pubrnd}}^*(i)$ described in Figure 5.25. The simulator runs a local copy of ${\Pi_{pubrnd}}^*$.

Simulatability Generation of random shares of honest parties is simulated exactly in the same way as in Proposition 5.3, and the only difference is that the resulting randomness r is broadcast to all parties, so we do not repeat here the proof that $\mathcal{S}_i$ simulates sharing the same r that was given by ${\mathcal{F}_{pubrnd}}^*$. In addition, it needs to simulate the checks $r^k_j = r'^k_j$.

- If $j \notin \mathcal{C}$, he would never choose such $r^k_j \neq r'^k_j$. Substituting the messages with something else for $j \notin \mathcal{C}$ cannot happen by definition of $\mathcal{F}_{transmit}$, .
- For $j \in \mathcal{C}$, r^{*k}_j are given to $\mathcal{S}_i$ by $\mathcal{A}^L$ (or generated by $\mathcal{S}_i$ itself if $\mathcal{A}^L$ chose $r^{*k}_j = \top$), so the check is easy to simulate.

Correctness ${\mathcal{F}_{pubrnd}}^*$ outputs r to $P_{p(id)}$. $\mathcal{S}_i$ generates r^k_j of $j \notin \mathcal{C}$ in such a way that $r = \sum_j \mathsf{declassify}(r^k_j)_{k \in \mathcal{H}}$, so this value is the same in both worlds. □

The protocol $\Pi_{pubrnd}{}^*$ works with unique identifiers id, encoding the bit length $m(id)$ of the randomness.

• **Initialization:** On input $(\mathsf{init}, \hat{m})$, assign the mapping $m \leftarrow \hat{m}$. For all $id \in \mathsf{Dom}(m)$, define mappings s, r, and f, such that $s(id_k^j) \leftarrow j, r(id_k^j) = f(id_k^j) \leftarrow k$ for all $id \in \mathsf{Dom}(p), j, k \in [n]$. In addition, define $s(id_k^{\mathsf{bc}j}) \leftarrow j$, for broadcasts. Send (init, s, r, f) to $\mathcal{F}_{transmit}{}^*$.

• **Randomness commitment:** On input (rnd, id):

1. Each party P_j, $j \neq p(id)$, generates a random value $r_j \in \mathbb{Z}_{2^{m(id)}}$, shares it to $(r_j^k)_{k\in[n]}$, writes $comm[id] \leftarrow (r_j^k)_{k\in[n]}$, and sends $(\mathsf{transmit}, id_k^j, r_j^k)$ to $\mathcal{F}_{transmit}{}^*$ for all $k \in [n]$.
2. P_j sends $(\mathsf{broadcast}, id_k^{\mathsf{bc}j}, r_j^k)$ to $\mathcal{F}_{transmit}{}^*$ for all $k \in [n]$.
3. Upon receiving $(id_k^{\mathsf{bc}j}, r_j^k)$ and $(id_k^j, r_j'^k)$ from $\mathcal{F}_{transmit}{}^*$, P_k checks if $r_j^k = r_j'^k$, and if the shares are consistent. If the check fails, use $\mathcal{F}_{transmit}{}^*$ to broadcast a complaint.
4. If a party P_i gets no complaints, it outputs $r = \sum_{j=1}^n \mathsf{declassify}(r_j^k)_{k\in[n]}$ to $\mathcal{Z}$.

• **Cheater detection:** At any time when $(\mathsf{cheater}, k)$ comes from $\mathcal{F}_{transmit}{}^*$, each party outputs $(\mathsf{cheater}, k)$ and assigns $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 5.24: The protocol $\Pi_{pubrnd}{}^*$

• **Initialization** $\mathcal{S}_i$ gets $(\mathsf{init}, \hat{m})$ from $\mathcal{F}_{pubrnd}{}^*$. It simulates the initialization of $\mathcal{F}_{transmit}{}^*$.

• **Randomness generation and sharing:** On input (rnd, id):

1. Similarly to $\mathcal{S}_{rnd}{}^*$, since the shares need to be distributed strictly *before* the broadcasts, $\mathcal{S}_i$ is able to adjust the shares of honest parties to the shares of corrupted parties given by $\mathcal{A}^L$. In addition to simulating cheap transmissions of $\mathcal{F}_{transmit}{}^*$, $\mathcal{S}_i$ simulates the broadcasts, where the broadcast shares r_j^k for $j \notin \mathcal{C}$ are exactly those that $\mathcal{S}_i$ has used in $r = \sum_{j=1}^n \mathsf{declassify}(r_j^k)_{k\in[n]}$. For $j \in \mathcal{C}$, $\mathcal{A}^L$ may say to $\mathcal{S}_i$ that some corrupted party refuses to send a message, or that it presents a complaint. In this case, $\mathcal{S}_i$ sends stop to $\mathcal{F}_{pubrnd}$.
2. After $\mathcal{S}_i$ has simulated all the broadcasts, it checks if any party P_k for $k \notin \mathcal{C}$ could have received $r_j^k \neq r_j'^k$. If the check fails for at least one P_k, it sends stop to $\mathcal{F}_{pubrnd}$ and also to each S_i^H that simulate sending a complaint using $\mathcal{F}_{transmit}{}^*$.

• **Cheater detection:** At any time when the $(\mathsf{cheater}, k)$ should be output by $\mathcal{F}_{transmit}{}^*$, $\mathcal{S}_i$ sends $(\mathsf{cheater}, k)$ to $\mathcal{F}_{pubrnd}{}^*$.

Figure 5.25: The simulator $\mathcal{S}_{pubrnd}{}^*(i)$

The Precomputed Tuple Generation Protocol $\Pi_{pre}{}^*$

An important change that we need to introduce into $\mathcal{F}_{pre}$ is, that we may no longer trust the prover to generate its own randomness, since it may choose it in a bad way, allowing to leak information to honest parties. For all types of tuples, we let all the basic non-correlated randomness be generated by $\mathcal{F}_{commit}^{weak}{}^*$, and $P_{p(id)}$ only helps the parties to compute non-trivial values that are uniquely determined by the basic randomness. The ideal functionality $\mathcal{F}_{pre}{}^*$ is given in Figure 5.26.

For the multiplication triples, the basic randomness is made up by the values a and b, from which $c = a \cdot b$ is computed. For the trusted bits, the basic randomness are actually the bits themselves, but the parties are able to collaboratively generate them only in $\mathbb{Z}_2$. First, the bits a_k are generated uniformly over $\mathbb{Z}_2$ using $\mathcal{F}_{commit}^{weak}{}^*$. The task of the verifier is to convert these bits into b_k committed in $\mathbb{Z}_{2^{m(id)}}$. In addition to proving the binariness of b_k, it should be proven that $a_k = b_k \pmod 2$. During the pairwise verification, the bit $c_{k,k'}$ is no longer chosen by the prover, but is computed by all parties as $c_{k,k'} =: \hat{a}_{k,k'} = a_k - a_{k'}$ in $\mathbb{Z}_2$.

The protocol $\Pi_{pre}{}^*$ implementing $\mathcal{F}_{pre}{}^*$ in WCP model is given in Figure 5.27-5.28. Similarly to Π_{pre} of Figure 4.27, we build it on top of a shared subroutine $\underline{\mathcal{F}_{commit}^{weak}{}^*}$, and we put on $\mathcal{Z}$ the same restriction, that the identifiers generated inside $\mathcal{F}_{pre}{}^*$ cannot be accessed externally by $\mathcal{Z}$.

Proposition 5.7. Let t be the upper bound on active coalition size. Assuming $t < n/2$, if $\mu > 1 + \eta/\log N$ and $\kappa > \max\{(N^{1/\mu} + 1)\eta, N^{1/\mu} + \mu - 1\}$, where N is the total number of generated tuples, the protocol $\Pi_{pre}{}^* \underline{\mathcal{F}_{commit}^{weak}{}^*}$-$t$-GWCP realizes $\mathcal{F}_{pre}{}^*$ in $\mathcal{F}_{commit}^{weak}{}^*$-hybrid model with correctness error $\varepsilon < 2^{\eta}$, and simulation error 0.

Proof. The outline of the simulator $\mathcal{S}_{pre}{}^*$ is analogous to $\mathcal{S}_{pre}$ given in Figure 4.29, and only the ports between the simulator and the adversary are different. The details of tuple generation have changed.

Simulatability. The simulator will need to generate some non-trivial values during the openings of the cut-and-choose and the pairwise verification. First, it simulates the generation of the basic components of all $\mu \cdot u(id) + \kappa$ tuples using $\mathcal{F}_{commit}^{weak}{}^*$. For $p(id) \notin \mathcal{C}$, it computes the remaining correlated components in such a way that all these $(\mu - 1)u(id) + \kappa$ tuples are valid. For $p(id) \in \mathcal{C}$, $\mathcal{A}^L$ may cheat with the remaining components of all $\mu \cdot u(id) + \kappa$ tuples.

Up to verifying first $\mu - 1$ tuples, the simulation is analogous to $\mathcal{S}_{pre}$, and it is rather trivial, since all the additional tuples are generated by $\mathcal{S}_i$ itself. The most interesting is the last, μ-th iteration. Let k be the index of the tuple that will be finally output and is not known to $\mathcal{S}_i$ (in general), and let k' be the index of the tuple against which the k-th tuple is being verified.

$\mathcal{F}_{pre}{}^*$ works with unique identifiers id, encoding a bit size $m(id)$ of the ring in which the tuples are committed, the party $p(id)$ that gets all the shares, and the number $u(id)$ of tuples to be generated. It stores an array $comm$ of already generated triple shares. It uses a shared subroutine $\underline{\mathcal{F}_{commit}^{weak}{}^*}$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{u}, \hat{p})$ from all (honest) parties, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{u}) = \mathsf{Dom}(\hat{p})$, assign the functions $m \leftarrow \hat{m}$, $u \leftarrow \hat{u}$, $p \leftarrow \hat{p}$. Deliver $\hat{m}$,$\hat{u}$,$\hat{p}$ to all ${\mathcal{A}_S}_i^H$.

• **Trusted bits:** On input (bit, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate a vector of random bits $\vec{b} \xleftarrow{\$} \mathbb{Z}_2^{u(id)}$.
2. Output $\vec{b}$ to ${\mathcal{A}_S}_{c(p(id))}^H$.
3. Wait until ${\mathcal{A}_S}^L$ inputs $\top$. Assign $comm[id] \leftarrow \vec{b}$. Output $\vec{b}$ to $P_{p(id)}$.
4. For all $k \in [u(id)]$, output $\underline{(\mathsf{commit}, id_0^k, b_k)}$ to $P_{p(id)}$, and $\underline{(\mathsf{commit}, id_0^k)}$ to each other (honest) party. These messages will be delivered to $\underline{\mathcal{F}_{commit}^{weak}{}^*}$.

• **Multiplication triples:** On input (triple, id) from all (honest) parties, if $comm[id]$ exists, then do nothing. Otherwise:

1. Generate $\vec{a} \xleftarrow{\$} \mathbb{Z}_{m(id)}^{u(id)}$, $\vec{b} \xleftarrow{\$} \mathbb{Z}_{m(id)}^{u(id)}$. Compute elementwise $\vec{c} = \vec{a} \cdot \vec{b}$.
2. Output $(\vec{a}, \vec{b}, \vec{c})$ to ${\mathcal{A}_S}_{c(p(id))}^H$.
3. Wait until ${\mathcal{A}_S}^L$ inputs $\top$. Assign $comm[id] = (\vec{a}, \vec{b}, \vec{c})$. Output $(\vec{a}, \vec{b}, \vec{c})$ to $P_{p(id)}$.
4. For all $k \in [u(id)]$, output $\underline{(\mathsf{commit}, id_0^k, a_k)}$, $\underline{(\mathsf{commit}, id_1^k, b_k)}$, $\underline{(\mathsf{commit}, id_2^k, c_k)}$ to $P_{p(id)}$, and $\underline{(\mathsf{commit}, id_0^k)}$, $\underline{(\mathsf{commit}, id_1^k)}$, $\underline{(\mathsf{commit}, id_2^k)}$ to each other (honest) party. These messages will be delivered to $\underline{\mathcal{F}_{commit}^{weak}{}^*}$.

• **Stopping:** At any time, on input (stop, id) from ${\mathcal{A}_S}^L$, stop the functionality and output $(id, \perp)$ to all parties.

Figure 5.26: Ideal functionality $\mathcal{F}_{pre}{}^*$

In $\Pi_{pre}{}^*$, each party works with unique identifiers id, encoding the bit size $m(id)$ of the ring in which the tuples are committed, the party $p(id)$ that gets the tuples, and the number $u(id)$ of tuples to be generated. $\Pi_{pre}{}^*$ uses $\mathcal{F}_{pubrnd}{}^*$ as a subroutine, and $\underline{\mathcal{F}_{commit}^{weak}{}^*}$ as a shared subroutine. The parameters μ and κ depend on the security parameter. Let λ be the number of bits in the randomness generator seed.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{u}, \hat{p})$ from $\mathcal{Z}$, where $\mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{u}) = \mathsf{Dom}(\hat{p})$, each party assigns the functions $m \leftarrow \hat{m}$, $u \leftarrow \hat{u}$, $p \leftarrow \hat{p}$. For each id, it defines the identifiers id_i^k for $k \in [\mu \cdot u(id) + \kappa]$, and $i \in [v]$, where $v = 2$ for trusted bits, and $v = 3$ for triples. It defines $\tilde{m}(id_i^k) \leftarrow m(id)$ (the exception is $\tilde{m}(id_0^k) \leftarrow 2$ for trusted bits), $\tilde{p}(id_i^k) \leftarrow p(id)$ for all i,k, and $\bar{m}(\mathsf{k}) = 2^\lambda$, where k is some fixed constant. It sends:

- $(\mathsf{ext}, \tilde{m}, \tilde{p})$ to $\underline{\mathcal{F}_{commit}^{weak}{}^*}$;
- $(\mathsf{init}, \bar{m})$ to $\mathcal{F}_{pubrnd}{}^*$.

• **Stopping:** If at any time $(\mathsf{cheater}, k)$ or $(id, \perp)$ come from $\underline{\mathcal{F}_{commit}^{weak}{}^*}$ or $\mathcal{F}_{pubrnd}{}^*$, output $(id, \perp)$ to $\mathcal{Z}$.

Figure 5.27: The protocol $\Pi_{pre}{}^*$ (initialization, stopping)

• **Multiplication triples:** On input (triple, id):

1. Each party delivers (rnd, id_0^k) and (rnd, id_1^k) to $\underline{\mathcal{F}_{commit}^{weak}}^*$ for all $k \in [\mu \cdot u(id) + \kappa]$.
2. Upon receiving (id_0^k, a_k) and (id_1^k, b_k) from $\underline{\mathcal{F}_{commit}^{weak}}^*$, $P_{p(id)}$ computes $c_k = a_k \cdot b_k$ and sends $(\mathsf{commit}, id_2^k, c_k)$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$. All other parties send $(\mathsf{commit}, id_2^k)$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$.
3. The parties send $(\mathsf{pubrnd}, \mathsf{k})$ to $\mathcal{F}_{pubrnd}$, getting back a randomness seed that they use to agree on a random permutation π of tuple indices. For $k \in [\kappa]$, each party sends $(\mathsf{weak\_open}, id_0^{\pi_k})$, $(\mathsf{weak\_open}, id_1^{\pi_k})$, $(\mathsf{weak\_open}, id_2^{\pi_k})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$, getting back (a_k, b_k, c_k). If the opening fails, or $c_k \neq a_k \cdot b_k$, then output $(id, \perp)$.
4. Taking the next $2 \cdot u(id)$ entries of π, the parties partition the corresponding triples into pairs. Such pairwise verification is repeated $\mu - 1$ times with the same $u(id)$ triples, each time taking the next $u(id)$ indices from π.

   For each pair (k, k'), let us denote $(id^a, id^b, id^c) = (id_0^k, id_1^k, id_2^k)$, $(id^{a'}, id^{b'}, id^{c'}) = (id_0^{k'}, id_1^{k'}, id_2^{k'})$, $(id^{\hat{a}}, id^{\hat{b}}, id^{\hat{c}}) = (id_0^{k,k'}, id_1^{k,k'}, id_2^{k,k'})$.

   (a) Each party sends $(id^{\hat{a}} = id^a - id^{a'})$, $(id^{\hat{b}} = id^b - id^{b'})$, and then $(\mathsf{weak\_open}, id^{\hat{a}})$, $(\mathsf{weak\_open}, id^{\hat{b}})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$, getting back $\hat{a}$ and $\hat{b}$ respectively.

   (b) Each party then sends $(id^{\hat{c}} = \hat{a} \cdot id^b + \hat{b} \cdot id^{a'} + id^{c'} - id^c)$ and $(\mathsf{weak\_open}, id^{\hat{c}})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$. If $\underline{\mathcal{F}_{commit}^{weak}}^*$ returns $\hat{c} \neq 0$, output $(id, \perp)$.
5. Let $\vec{id}$ be the vector of the identifiers of the remaining $u(id)$ triples in $\underline{\mathcal{F}_{commit}^{weak}}^*$. For $id' \in \vec{id}$, $P_{p(id)}$ outputs $comm[id']$.

• **Trusted bits:** On input (bit, id):

1. Each party delivers (rnd, id_0^k) to $\underline{\mathcal{F}_{commit}^{weak}}^*$ for all $k \in [\mu \cdot u(id) + \kappa]$.
2. Upon receiving (id_0^k, a_k) from $\underline{\mathcal{F}_{commit}^{weak}}^*$, where $a_k \in \mathbb{Z}_2$, the party $P_{p(id)}$ takes $b_k = a_k$ in $\mathbb{Z}_{2^{m(id)}}$, and sends $(\mathsf{commit}, id_1^k, b_k)$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$. All other parties send $(\mathsf{commit}, id_1^k)$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$.
3. The parties send $(\mathsf{pubrnd}, \mathsf{k})$ to $\mathcal{F}_{pubrnd}$, getting back a randomness seed that they use to agree on a random permutation π of tuple indices. For $k \in [\kappa]$, each party sends $(\mathsf{weak\_open}, id_0^{\pi_k})$ and $(\mathsf{weak\_open}, id_1^{\pi_k})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$, getting back a_k and b_k respectively. If the opening fails, or $a_k \neq b_k$, then output $(id, \perp)$.
4. Taking the next $2 \cdot u(id)$ entries of π, the parties partition the corresponding bits into pairs.

   For each pair (k, k'), let us denote $(id^a, id^b) = (id_0^k, id_1^k)$, $(id^{a'}, id^{b'}) = (id_0^{k'}, id_1^{k'})$, $(id^{\hat{a}}, id^{\hat{b}}) = (id_0^{k,k'}, id_1^{k,k'})$.

   (a) Each party sends $(id^{\hat{a}} = id^a - id^{a'})$ followed by $(\mathsf{weak\_open}, id^{\hat{a}})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$, getting back $\hat{a}$.

   (b) If $\hat{a} = 0$, each party sends $(id^{\hat{b}} = id^b - id^{b'})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$. If $\hat{a} = 1$, each party sends $(id^{\hat{b}} = 1 - id^b - id^{b'})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$.

   (c) Each party then sends $(\mathsf{weak\_open}, id^{\hat{b}})$ to $\underline{\mathcal{F}_{commit}^{weak}}^*$. If $\underline{\mathcal{F}_{commit}^{weak}}^*$ returns $\hat{b} \neq 0$, output $(id, \perp)$.
5. Let $\vec{id}$ be the vector of the identifiers of the remaining $u(id)$ bits in $\underline{\mathcal{F}_{commit}^{weak}}^*$. For $id' \in \vec{id}$, $P_{p(id)}$ outputs $comm[id']$.

Figure 5.28: The protocol $\Pi_{pre}{}^*$ (tuple generation)

- **Multiplication Triples:** The generation of initial random components is reduced to ${\mathcal{F}_{commit}^{weak}}^*$, For $p(id) \in \mathcal{C}_i$, $\mathcal{S}_i$ simulates it in such a way that the last $u(id)$ tuples that will be finally accepted are those chosen by ${\mathcal{F}_{pre}}^*$. Let x and x' be two values whose differences are opened during the pairwise check. Since both x and x' are assumed to come from ${\mathcal{F}_{commit}^{weak}}^*$, and x' has not been opened yet, all published differences $\hat{x} = x - x'$ are distributed uniformly, and hence can be generated by $\mathcal{S}_i$ for $p(id) \notin \mathcal{C}_i$. If $\mathcal{A}^L$ wants to cheat and open $\hat{x}^* \neq \top$, then opening $\hat{x}^*$ is simulated by $\mathcal{S}_i$.

- **Trusted bits:** We have modified our protocol in such a way that the bits b_k need to be additionally verified against $a_k \in \mathbb{Z}_2$. For this, a linear combination $\hat{a}_{k,k'} = a_k - a_{k'}$ needs to be opened. Similarly to the multiplication triple case, $a_{k'}$ serves as a mask for a_k.

In addition, $\mathcal{S}_i$ should correctly simulate the cut-and-choose openings, and the finally opened alleged zeroes. Since all basic components are either generated by $\mathcal{S}_i$ itself, or are given to it by ${\mathcal{F}_{pre}}^*$, it is able to check whether the additional component c for the triple (a, b, c), and b for the trusted bit pair (a, b), are computed properly by $\mathcal{A}^L$.

Correctness. For $p(id) \notin \mathcal{C}$, the finally remaining $u(id)$ tuples are exactly those that are generated by ${\mathcal{F}_{pre}}^*$. For $p(id) \in \mathcal{C}$, these $u(id)$ tuples are all generated by $\mathcal{A}^L$. We need to show that, if the final $u(id)$ tuples are accepted for $p(id) \in \mathcal{C}$, then they are all valid, except with negligible probability.

First of all, we show that, if the tuple with the index k' is valid, then the pairwise check passes only if the tuple k is also valid.

- **Multiplication Triples:** The only difference of ${\Pi_{pre}}^*$ from Π_{pre} is that the generation of initial random components is reduced to ${\mathcal{F}_{commit}^{weak}}^*$, so that the initial tuple components are indeed random, as ${\mathcal{F}_{pre}}^*$ requires. The correctness of the other components that are computed from them is ensured in the same way as in Π_{verify}.

- **Trusted bits:** The additional requirement of ${\mathcal{F}_{pre}}^*$ is that the bits b_i should be truly random. Since $1 - b_{k'} - b_k = 0$ should hold for $a_{k'} \neq a_k$, and $b_k - b_{k'}$ should hold for $a_{k'} = a_k$, these checks ensure that the bits a_k differ from $a_{k'}$ in the same way as b_k differ from $b_{k'}$. Assuming that $(a_{k'}, b_{k'})$ is a valid tuple, i.e. $a_{k'} = b_{k'}$, we get $a_k = b_k$.

We have shown that the only possibility for the prover to cheat is to put two invalid tuples into the same pair. For the $\mu - 1$ pairwise checks, the finally accepted invalid tuple should be paired with some other invalid tuple on each iteration. As we have shown in Lemma 4.6, for sufficiently large μ and κ, this happens only with a negligible probability. □

Table 5.3: Number of tuple bits involved in different steps (ring cardinality 2^m)

	$op(x,m)$				
x	$\mathsf{nb}_{\mathsf{tpl}_1}$	$\mathsf{nb}_{\mathsf{tpl}_1}$	$\mathsf{nb}_{\mathsf{tpl}_2}$	$\mathsf{nb}_{\mathsf{op}_1}$	$\mathsf{nb}_{\mathsf{op}_2}$
bit	$1+m$	1	m	1	m
triple	$3m$	$2m$	m	$2m$	m

Observation 5.4. From the description of ${\Pi_{pre}}^*$, we can extract the total number of bits broadcast in different steps of the tuple generation:

- the number of bits $\mathsf{nb}_{\mathsf{tpl}}(T)$ in a single tuple of type T;
- the number of bits $\mathsf{nb}_{\mathsf{tpl}_1}(T)$ generated using ${\mathcal{F}_{commit}^{weak}}^*$;
- the number of bits $\mathsf{nb}_{\mathsf{tpl}_2}(T) = \mathsf{nb}_{\mathsf{tpl}}(T) - \mathsf{nb}_{\mathsf{tpl}_1}(T)$ that are computed by the prover;
- the numbers $\mathsf{nb}_{\mathsf{op}_1}(T)$ and $\mathsf{nb}_{\mathsf{op}_2}(T)$ of tuple bits opened in the pairwise check, where $\mathsf{nb}_{\mathsf{op}_1}(T)$ bits are opened before the last $\mathsf{nb}_{\mathsf{op}_2}(T)$ bits.

For Π_{pre}, we have $\mathsf{nb}_{\mathsf{tpl}_1}(T) = 0$ and $\mathsf{nb}_{\mathsf{tpl}_2}(T) = \mathsf{nb}_{\mathsf{tpl}}(T)$. Compared to Table 4.4, the costs have changed for trusted bits, and there are now also some new tuples. The new costs are given in Table 5.3.

The cost of preprocessing can be computed similarly to the value proven in Lemma 4.5, and the difference is that now not all $\mathsf{nb}_{\mathsf{tpl}}(T)$ bits are transmitted to each party, but $\mathsf{nb}_{\mathsf{tpl}_1}(T)$ need to be generated by ${\mathcal{F}_{commit}^{weak}}^*$, and only $\mathsf{nb}_{\mathsf{tpl}_2}(T)$ are transmitted. Taking into account the cost of randomness generation of Table 5.1, and denoting $M := \mu N + \kappa$ the cost can be rewritten as

$$\begin{aligned}\mathsf{prc}_T^N \;=\; & (\mathsf{tr}_{\mathsf{sh}_n\cdot\lambda}^{\otimes nt} \oplus \mathsf{bc}_{\mathsf{sh}_n\cdot\lambda}^{\otimes n}) \\ & \otimes (\mathsf{tr}_{M\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}_1}T)}^{\otimes n(t+1)} \oplus \mathsf{fwd}_{M\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}_1}T)}^{\otimes n(t+1)} \oplus \mathsf{tr}_{M\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}_1}T)}^{\otimes n(t+1)}) \\ & \oplus (\mathsf{bc}_{\kappa\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}}T)}^{\otimes n} \otimes \mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_1}T)}^{\otimes n} \otimes \mathsf{bc}_{(\mu-1)N\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{op}_2}T)}^{\otimes n}) \;.\end{aligned}$$

Since $t+1 < n$, and instead of n transmissions of $(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}}T)$ bits there are now $2n^2$ transmissions and n^2 forwardings of $(\mu N+\kappa)\cdot\mathsf{sh}_n\cdot(\mathsf{nb}_{\mathsf{tpl}_1}T)$ bits each, which are as expensive as $3n^2$ transmissions, the rough multiplicative overhead of ${\Pi_{pre}}^*$ compared to Π_{pre} is upper bounded by $3n$. We do not take into account preprocessing phase of ${\mathcal{F}_{transmit}}^*$, since ${\mathcal{F}_{pre}}^*$ is allowed to fail, and the steps involving tuple generation do not need to support the expensive mode of ${\mathcal{F}_{transmit}}^*$ on which ${\Pi_{commit}^{weak}}^*$ is built. The number of rounds remains the same.

$\mathcal{F}_{verify}{}^*$ works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ committed to $comm[id]$, the function $f(id)$ to verify, and the input identifiers $\vec{xid}(id)$ on which $f(id)$ should be verified w.r.t. the output identified by id. It also encodes the randomness $r(id)$ that is generated during the initialization, and will be used later for the randomness commitment. The messages are first stored in an array $sent$ before the sender and the receiver get finally committed to them.

• **Initialization:** On input $(\mathsf{init}, \hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'})$ from all (honest) parties, where $\hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'}$ are defined over the same domain, assign $f \leftarrow \hat{f}$, $\vec{xid} \leftarrow \hat{\vec{xid}}$, $p \leftarrow \hat{p}$, $p' \leftarrow \hat{p'}$. For all $id \in \mathsf{Dom}(f)$, generate a fresh randomness $r(id)$ in $\mathbb{Z}_{2^m}$, where $\mathbb{Z}_{2^m}$ is the range of $f(id)$. For $p(id) \in \mathcal{C}$, deliver $r(id)$ to $\mathcal{A}_S{}^H_{c(p(id))}$. Deliver $(\mathsf{init}, f, \vec{xid}, p, p')$ to all $\mathcal{A}_S{}^H_i$. If $\mathcal{A}_S{}^L$ responds with (stop), output $\perp$ to all parties.

• **Input Commitment:** On input $(\mathsf{commit\_input}, id, x)$ from $P_{p(id)}$, and $(\mathsf{commit\_input}, id)$ from all (honest) parties, if $comm[id]$ is not defined yet, assign $comm[id] \leftarrow x$. If $p(id) \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S{}^L$.

• **Message Commitment:** On input $(\mathsf{send\_msg}, id, x)$ from $P_{p(id)}$ and $(\mathsf{send\_msg}, id)$ from all (honest) parties, output x to $\mathcal{A}_S{}^H_{p'(id)}$. If $p(id) \in \mathcal{C}$, then x is chosen by $\mathcal{A}_S{}^L$. Output x to $P_{p'(id)}$. Assign $sent[id] \leftarrow x$.
On input $(\mathsf{commit\_msg}, id)$ from all (honest) parties, check if $sent[id]$ and $comm[id]$ are defined. If $sent[id]$ is defined, and $comm[id]$ is not defined, assign $comm[id] \leftarrow sent[id]$. If both $p(id), p'(id) \in \mathcal{C}$, assign $comm[id] \leftarrow x^*$, where x^* is chosen by $\mathcal{A}_S{}^L$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$ from $P_{p(id)}$, and $(\mathsf{commit\_rnd}, id)$ from all (honest) parties, check if $comm[id]$ exists. If it does, then do nothing. Otherwise, assign $comm[id] \leftarrow r(id)$.

• **Verification:** On input (verify, id) from all (honest) parties, if $comm[id]$ and $comm[i]$ have been defined for all $i \in \vec{xid}(id)$, take $\vec{x} \leftarrow (comm[i])_{i \in \vec{xid}(id)}$ and $y \leftarrow comm[id]$. For $f \leftarrow f(id)$, compute $y' \leftarrow f(\vec{x})$. If $y' - y = 0$, output $(id, 1)$ to each party. Otherwise, output $(id, 0)$ to each party. Output the difference $y' - y$, to all adversaries $\mathcal{A}_S{}^H_i$.

• **Cheater detection:** On all inputs involving id, if $p(id) \in \mathcal{C}$, $\mathcal{A}_S$ may input $(\mathsf{cheater}, p(id))$. In this case, $comm[id]$ is not assigned. On input $(\mathsf{send\_msg}, id)$, $sent[id]$ is not assigned. If no $(\mathsf{cheater}, p(id))$ comes, then each commitment ends up outputting $(\mathsf{confirmed}, id)$ to each party.
On each input $(\mathsf{cheater}, k)$ from $\mathcal{A}_S$ for $k \in \mathcal{C}$, output $(\mathsf{cheater}, k)$ to each party. Assign $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$, $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$.

Figure 5.29: Ideal functionality $\mathcal{F}_{verify}{}^*$

The Verification Protocol Π_{verify}

Figure 5.29 depicts the functionality $\mathcal{F}_{verify}{}^*$, defined similarly to the UC functionality $\mathcal{F}_{verify}$ given in Figure 4.30. As in $\mathcal{F}_{commit}^{weak}{}^*$, all the randomness that needs to be committed will be generated already during the initialization. On input $(\mathsf{commit\_rnd}, id)$, the parties will just write that randomness into $comm[id]$.

If we use the UC protocol Π_{verify} of Figure 4.31 without special modifications, just building it on top of $\mathcal{F}_{transmit}{}^*$ and $\mathcal{F}_{commit}^{weak}{}^*$, we will be able to WCP-realize $\mathcal{F}_{verify}{}^*$. It will be sufficient to implement the verification for a *covert* adversary, since if no party attempts to cheat due to being detected, it will always commit y such that $f(\vec{x}) - y = 0$.

In $\Pi_{verify}{}^*$, each party works with unique identifiers id, encoding the party indices $p(id)$ and $p'(id)$ committed to $comm[id]$, the function $f(id)$ to verify, and the identifiers $\vec{xid}(id)$ of the inputs on which $f(id)$ should be verified w.r.t. the output identified by id. It also encodes the randomness $r(id)$ that will be precomputed by the parties during the initialization, and will later be used for randomness commitment. The prover stores the committed values in a local array $comm$. The verifiers store the helpful values published by the verifier in an array $pubv$. The messages are stored by the sender and the receiver in a local array $sent$ before they finally get committed to these messages. $\Pi_{verify}{}^*$ uses $\mathcal{F}_{transmit}{}^*$, $\mathcal{F}_{commit}^{weak}{}^*$, $\mathcal{F}_{pre}{}^*$ as subroutines.

• **Initialization:** On input $(\mathsf{init}, \hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p'})$, where domains of the mappings $\hat{f}, \hat{\vec{xid}}, \hat{p}, \hat{p}'$ are the same, initialize $comm$ and $sent$ to empty arrays. Assign $f \leftarrow \hat{f}$, $\vec{xid} \leftarrow \hat{\vec{xid}}$, $p \leftarrow \hat{p}$, $p' \leftarrow \hat{p'}$. Initialize $\mathcal{F}_{transmit}{}^*$, $\mathcal{F}_{commit}^{weak}{}^*$, and $\mathcal{F}_{pre}{}^*$ in the same way as it is done in Π_{verify} of Figure 4.31.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$, each party sends (rnd, id) to $\mathcal{F}_{commit}^{weak}{}^*$.

• **Other inputs:** On all other inputs, the parties behave in the same way as defined in Π_{verify}.

Figure 5.30: The protocol $\Pi_{verify}{}^*$ (modifications compared to Π_{verify})

If the adversary is *active*, then $\mathcal{F}_{verify}{}^*$ will not provide reasonable security guarantees in WCP model. We will discuss this problem in Section 5.4.4, when we construct verifiable computation for active adversaries.

Since the randomness generation is now done by $\mathcal{F}_{commit}^{weak}{}^*$, and not $\mathcal{F}_{rnd}$ as it was in Π_{verify}, we formally need to modify the definition of Π_{verify}. The resulting protocol $\Pi_{verify}{}^*$ is given in Figure 5.30. We only need to re-define the actions of parties on input $(\mathsf{commit\_rnd}, id)$. The behaviour of parties on all other inputs remains the same.

Proposition 5.8. Let t be the upper bound on covert coalition size. Assuming $t < n/2$, and a *covert* adversary, the protocol $\Pi_{verify}{}^*$ t-WCP-realizes $\mathcal{F}_{verify}{}^*$ in $\mathcal{F}_{transmit}{}^*$-$\mathcal{F}_{commit}^{weak}{}^*$-$\mathcal{F}_{pre}{}^*$-hybrid model.

Proof. We use the simulator $\mathcal{S}_i = \mathcal{S}_{verify}{}^*(i)$ described in Figure 5.31. The simulator runs a local copy of $\Pi_{verify}{}^*$, and local copies of $\mathcal{F}_{transmit}{}^*$, $\mathcal{F}_{commit}^{weak}{}^*$, $\mathcal{F}_{pre}{}^*$.

Simulatability. During the initialization, $\mathcal{S}_i$ only needs to simulate the initialization of the subroutines. For $k \in \mathcal{C}_i$, it should deliver to $\mathcal{A}_i^H$ the randomness that $\mathcal{F}_{commit}^{weak}{}^*$ would output to $\mathcal{A}_i^H$. $\mathcal{S}_i$ gets this randomness from $\mathcal{F}_{verify}{}^*$.

During the commitments, $\mathcal{S}_i$ simulates $\mathcal{F}_{commit}^{weak}{}^*$; the inputs of dishonest parties for this functionality are provided by $\mathcal{A}^L$.

When the verification starts, $\mathcal{S}_i$ needs to simulate the broadcast, and it needs to generate the broadcast values of $p(id) \notin \mathcal{C}_i$ itself. Similarly to $\mathcal{S}_{verify}$ of Figure 4.34, all of these values are some private values hidden by a random mask.

After all the broadcasts and subsequent local operations on $\mathcal{F}_{commit}^{weak}{}^*$ (which do not require any interaction) are simulated, $\mathcal{S}_i$ simulates opening to each party

• **Initialization:** $\mathcal{S}_i$ gets $(\mathsf{init}, f, \vec{xid}, p, p')$, from ${\mathcal{F}_{verify}}^*$. First of all, $\mathcal{S}_i$ simulates initialization of ${\mathcal{F}_{commit}^{weak}}^*$. It gets randomness $r(id)$ from ${\mathcal{F}_{verify}}^*$, that is expected to be committed using ${\mathcal{F}_{commit}^{weak}}^*$ for $p(id) \in \mathcal{C}_i$. During the simulation of initialization of ${\mathcal{F}_{commit}^{weak}}^*$, $\mathcal{S}_i$ outputs to $\mathcal{A}_i^H$ the same values $r(id)$ that were given by ${\mathcal{F}_{verify}}^*$.
$\mathcal{S}_i$ simulates initializing and running ${\mathcal{F}_{pre}}^*$. It receives from ${\mathcal{F}_{pre}}^*$ the tuples for $p(id) \in \mathcal{C}_i$ and outputs them to $\mathcal{A}_i^H$. If its execution has not failed, then $\mathcal{A}_i^H$ expects that all (valid) tuples for all $p(id) \in [n]$ are copied to ${\mathcal{F}_{commit}^{weak}}^*$. If the execution fails, $\mathcal{S}_i$ sends (stop) to ${\mathcal{F}_{verify}}^*$.

• **Input Commitment:** $\mathcal{S}_i$ simulates sending (commit, id, x) and (commit, id) to ${\mathcal{F}_{commit}^{weak}}^*$. For $p(id) \in \mathcal{C}_i$, x is computed from the local view of $\mathcal{S}_i$. For $p(id) \in \mathcal{C}$, $\mathcal{S}_i$ obtains x^* from $\mathcal{A}^L$. It delivers x^* to ${\mathcal{F}_{verify}}^*$.

• **Message Commitment:** First of all, $\mathcal{S}_i$ simulates sending $(\mathsf{transmit}, id, m)$ to ${\mathcal{F}_{transmit}}^*$. Then $\mathcal{S}_i$ simulates sending $(\mathsf{mcommit}, id, id', m)$, $(\mathsf{mcommit}, id, id', m')$, and $(\mathsf{mcommit}, id, id')$ to ${\mathcal{F}_{commit}^{weak}}^*$, where m' is the message that $p'(id)$ has actually received, and id' is the identifier that corresponds to $p'(id)$ in ${\mathcal{F}_{commit}^{weak}}^*$. To do this simulation, $\mathcal{S}_i$ needs to know the bit b denoting $m = m'$. It takes $b = 1$ iff either $m^* \neq \top$ or $m'^* \neq \top$ was chosen by $\mathcal{A}^L$ (if both $m^*, m'^* \neq \top$, then take $b = (m^* = m'^*)$). If $m^* = m'^*$, then m^* is delivered to ${\mathcal{F}_{verify}}^*$.

• **Randomness Commitment:** On input $(\mathsf{commit\_rnd}, id)$, $\mathcal{S}_i$ sends (rnd, id) to ${\mathcal{F}_{commit}^{weak}}^*$.

• **Stoppings:** At any time, when ${\mathcal{F}_{transmit}}^*$, ${\mathcal{F}_{pre}}^*$, or ${\mathcal{F}_{commit}^{weak}}^*$ should output a message $(\mathsf{cheater}, k)$, $\mathcal{S}_i$ outputs $(\mathsf{cheater}, k)$ to ${\mathcal{F}_{verify}}^*$ and discards P_k from their local run of ${\Pi_{verify}}^*$, assigning $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$ and $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$.

• **Verification:** On input (verify, id), $\mathcal{S}_i$ decomposes $f(id)$ to basic operations $f_1, \ldots, f_N$, and defines the additional identifiers $id_i^{\mathsf{x}_k}$, $id_i^{\mathsf{y}_k}$, $id_i^{\mathsf{z}_k}$ as the honest parties do.

1. For $p(id) \in \mathcal{C}_i$, $\mathcal{S}_i$ computes all the intermediate values $comm[id_i^{\mathsf{x}_k}]$ and $comm[id_i^{\mathsf{y}_k}]$, uses them to compute $\hat{\vec{x}}$, and simulates broadcasting it. For $p(id) \notin \mathcal{C}_i$, $\mathcal{S}_i$ samples $\hat{\vec{x}}$ from appropriate uniform distribution, similarly to $\mathcal{S}_{verify}$ of Figure 4.34. For $p(id) \in \mathcal{C}$, $\mathcal{S}_i$ uses $\hat{\vec{x}}^*$ given by $\mathcal{A}^L$. If $\hat{\vec{x}}^* = \top$, it takes the $\hat{\vec{x}}$ that it computed itself.
   $\mathcal{S}_i$ simulates broadcasting $\hat{\vec{x}}^*$ and $\hat{\vec{x}}$ through $\mathcal{F}_{transmit}$. It writes $pubv[id_i^{type}] \leftarrow \hat{\vec{x}}$ for all honest parties receiving $\hat{\vec{x}}$. As $\mathcal{S}_{verify}$ does, for the trusted bits it additionally simulates sending the messages $(id_{i,k}^{\mathsf{bit}} = 1 - id_{i,k}^{\mathsf{bit}})$ to ${\mathcal{F}_{commit}^{weak}}^*$ (requiring no interaction) for which $c_k = 1$ was broadcast, as the honest parties do.
2. The local computation of the verifiers depends on f_i, and $\mathcal{S}_i$ just simulates using ${\mathcal{F}_{commit}^{weak}}^*$ to compute certain local operations. In the end, $\mathcal{S}_i$ needs to simulate opening to each party the alleged zero vector $\vec{z}$. For $p(id) \in \mathcal{C}_i$, $\mathcal{S}_i$ already knows all the values needed to compute $\vec{z}$. For $p(id) \notin \mathcal{C}_i$, $\mathcal{S}_i$ obtains the difference $f(\vec{x}) - y$ from ${\mathcal{F}_{verify}}^*$. It assigns $\vec{z}_{N+1} \leftarrow [f(\vec{x}) - y]$. For all other alleged zeroes z, it takes $z \leftarrow 0$ for $p(id) \notin \mathcal{C}$. For $p(id) \in \mathcal{C}$, the choice of z depends on the actions of $\mathcal{A}^L$. For basic operations, we have the following types of alleged zeroes:
   - $z = x' - x - r_x$ for some value x' that is broadcast by the prover. In the first round, $\mathcal{S}_i$ has already generated a uniformly distributed x', and $\mathcal{A}^L$ has chosen to broadcast x'^* instead of x'. $\mathcal{S}_i$ takes $z = x'^* - x'$.
   - $z = x - \sum_{k=1}^{m} 2^{k-1} y_k$, where for $k \in [m]$ a bit c_k was broadcast, denoting whether $y_k = b_k$ or $y_k = 1 - b_k$. $\mathcal{A}^L$ has chosen to broadcast c_k^* instead of c_k. $\mathcal{S}_i$ takes $z = \sum_{k=1}^{m} (2^{k-1}(c_k^* - c_k) \bmod 2)$.

   In the end, $\mathcal{S}_i$ simulates opening $\vec{z}$, and the final decisions of parties based on the opened $\vec{z}$.

Figure 5.31: The simulator ${\mathcal{S}_{verify}}^*(i)$

the alleged zero vector $\vec{z}$. If $p(id) \in \mathcal{C}_i$, then $\mathcal{S}_i$ already knows all the values needed to compute $\vec{z}$.

If $p(id) \notin \mathcal{C}_i$, then $\mathcal{S}_i$ obtains only the difference $f(\vec{x}) - y$ from ${\mathcal{F}_{verify}}^*$. However, it needs to simulate the alleged zeroes $\vec{z}_i$ of *each* intermediate basic function f_i. If $p(id) \notin \mathcal{C}$, it would have broadcast $\hat{\vec{x}}$ such that $z = 0$ for the remaining entries z of $\vec{z}$. It is more complex with $p(id) \in \mathcal{C} \setminus \mathcal{C}_i$. It may happen that some entries of $\vec{x}$ and $\vec{y}$ have not been chosen by $\mathcal{A}^L$, so $\mathcal{S}_i$ does not have enough information to simulate $\vec{z}$. Putting aside the alleged zeroes corresponding to the final answers $f(\vec{x}) - y$ (that $\mathcal{S}_i$ obtains from ${\mathcal{F}_{verify}}^*$), we have the following types of alleged zeroes:

- $z = x' - x - r_x$ for some value x' that has been broadcast by the prover. $\mathcal{S}_i$ has already generated a random x' and broadcast x'^*. Sampling x' by the simulator was not in contradiction with the view of $\mathcal{A}_i^H$ so far, as the randomness r_x was used the first time to mask x. Since $x' = x + r_x$ was assumed, and x'^* was actually broadcast, $\mathcal{A}_i^H$ now expects $z = x'^* - x - r_x = x'^* - x'$.

- $z = x - \sum_{k=1}^{m} 2^{k-1} y_k$, where for $k \in [m]$ a bit c_k was broadcast, denoting whether $y_k = b_k$ or $y_k = 1 - b_k$. The reasoning here is similar to $z = x' - x - r_x$, just all m bits are verified simultaneously, and all $\mathbb{Z}_2$ arithmetic has to be performed in $\mathbb{Z}_{2^m}$. Sampling c_k by the simulator was not in contradiction with the view of $\mathcal{A}_i^H$ so far, since the bit b_k comes from uniform distribution over $\{0, 1\}$. Since the verification would stop if $c_k \notin \{0, 1\}$ was broadcast, $\mathcal{A}^L$ could at most change the bit value c_k. If it happens that $c_k = c_k^*$, then the bit $z_k = 0$ is expected by $\mathcal{A}_i^H$. If $c_k = c_k^*$, then $c_k^* = 1$, so $\mathcal{A}_i^H$ assumes that y_k is a flipped x_k, so it should be $z_k = 1$.

Since the adversary is covert, we may assume that weak_open always succeeds (otherwise, open would be called, causing the cheater to be blamed), and so there will be no need to call open.

Correctness. The inputs / messages of $p(id) \notin [n]$ (modified by $\mathcal{A}^L$ for $p(id) \in \mathcal{C}$), and the randomness chosen by ${\mathcal{F}_{verify}}^*$ are all stored in ${\mathcal{F}_{commit}^{weak}}^*$. In addition, the precomputed tuples are also stored in the same ${\mathcal{F}_{commit}^{weak}}^*$ by definition of ${\mathcal{F}_{pre}}^*$. ${\mathcal{F}_{commit}^{weak}}^*$ may now be used as a black box, doing computation on all these commitments. It remains to prove that, if all these values are committed properly, then ${\Pi_{verify}}^*$ does verify the computation of $f(id)$ on input (verify, id).

It is easy to see that, if $\vec{z}_i = \vec{0}$ for the alleged zeroes produced by the basic function f_i, then f_i has been computed correctly with respect to the committed inputs and outputs on which it was verified, and $\hat{\vec{x}}_i$ has been computed correctly for f_i. The details of verifying each basic function are analogous to the Lemma 4.10,

so we do not repeat the proof here. If all f_i have been computed correctly, then so is their composition f. □

Proposition 5.9. Let M_c be the total number of bits sent in the execution phase of the original passively secure protocol. Compared to the UC protocol Π_{verify} built on top of Π_{commit}, Π_{rnd} and Π_{pre}, the protocol ${\Pi_{verify}}^*$ built on top of ${\Pi_{transmit}}^*$, ${\Pi^{weak}_{commit}}^*$ and ${\Pi_{pre}}^*$ has the following costs:

- *Preprocessing:* $X_{pre}^{\otimes 3n} \otimes (\mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 2n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M_c}^{\otimes 3n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 3n(t+1)})$, where X_{pre} is the cost of the preprocessing of Π_{verify}.
- *Execution:* No overheads in the cheap mode. $X_{exec}^{\otimes n}$ in the expensive mode, where X_{exec} is the cost of the expensive mode of the execution of Π_{verify}.
- *Postprocessing:* $X_{post}^{\otimes n}$, where X_{post} is the cost of either the cheap or the expensive mode of the postprocessing of Π_{verify}.

Proof. We justify the estimations given to complexities of different phases.

- *Preprocessing:* In this phase, by definition of Π_{verify}, the functionalities $\mathcal{F}_{transmit}$, $\mathcal{F}_{commit}$, and $\mathcal{F}_{pre}$ are initialized, and $\mathcal{F}_{pre}$ is executed to generate a certain number of tuples. The same holds for ${\Pi_{verify}}^*$ regarding ${\mathcal{F}_{transmit}}^*$, ${\mathcal{F}^{weak}_{commit}}^*$ and ${\mathcal{F}_{pre}}^*$. The protocols of WCP model incur the following overheads:
  - Initialization and execution of ${\Pi_{pre}}^*$ gives $3n$ multiplicative overhead compared to Π_{pre} by Observation 5.4.
  - Initialization of ${\mathcal{F}_{transmit}}^*$ supporting expensive mode requires additional randomness to be generated during preprocessing. ${\Pi_{verify}}^*$ uses ${\mathcal{F}_{transmit}}^*$ to transfer the M_c bits of the initial protocol. By Observation 5.2, the cost of this initialization is $(\mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M_c}^{\otimes 2n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 2n(t+1)})$.
  - Initialization of ${\mathcal{F}^{weak}_{commit}}^*$ requires additional shared randomness to be generated, and by Observation 5.3 its additional cost is $(\mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M_c}^{\otimes n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes n(t+1)}) \otimes (\mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M_c}^{\otimes 2n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 2n(t+1)}) = (\mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 2n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M_c}^{\otimes 3n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 3n(t+1)})$, assuming that the inputs M_x are not longer than the communication M_c.
- *Execution:* No overheads in the cheap mode. $X^{\otimes n}$ in the expensive mode by Observation 5.2, where X is the cost of the expensive mode of execution phase of Π_{verify}.

- *Postprocessing:* For commitments, the multiplicative overhead of the online phase is n by Observation 5.3, as the broadcast is n times more costly than n transmissions. In the expensive mode, the broadcast is ca. $1.5n$ times more costly than n transmissions, which is asymptotically the same (we used Table 4.1 for these estimations). □

5.4.4 Active Adversaries

In order to achieve active security, we could apply the verification after each protocol round. However, we cannot use $\Pi_{verify}{}^*$ anymore. Differently from the case of covert adversary, $\mathcal{S}_i$ is no longer able to simulate resolving the conflicts regarding alleged zero vector for a corrupted party, since an active adversary is not afraid of being caught. We discuss these problems in detail.

Non-zero alleged zero. One problem comes from opening the alleged zero vector $\vec{z}$ itself, before any conflict resolving takes place. In UC model, it could not leak information of an honest party since an honest prover does not allow to open anything except $\vec{z} = \vec{0}$. In the WCP model, the adversary may use $\vec{z}$ to leak sensitive information to some *honest* party. For example, if the parties are checking if P_k has correctly computed $3x = y$, then P_k may substitute y with 0 in the check $3x - y = 0$, so that $z = 3x$ becomes published. If x has been sent to P_k by some honest P_i, it may be unhappy about leaking $3x$ to everyone. Hence it is now questionable whether we should allow $\mathcal{F}_{verify}{}^*$ to open the differences $f(\vec{x}) - y$ to honest parties.

In a finite field $\mathbb{Z}_q$, the problem of of checking $z = 0$ could be solved by multiplicative hiding, checking $z \cdot r = 0$ for a uniformly distributed r. If $z \neq 0$, then product $z \cdot r$ is also distributed uniformly over the field, so only a single bit of information is leaked, whether $z = 0$ or not. This multiplication could be done again using precomputed multiplication triples over the corresponding field.

In a ring $\mathbb{Z}_{2^m}$, the simplest solution is to decompose each entry z_j of the alleged zero vector $\vec{z}$ of length ℓ to bits $(z_{j1}, \ldots, z_{jm})$ using trusted bits. The product $\prod_{j,i=1}^{\ell,m}(1 - z_{ji})$ in $\mathbb{Z}_2$ returns a single bit, denoting whether $\vec{z} = \vec{0}$. All the multiplications can be done using precomputed triples over $\mathbb{Z}_2$. As the result, we still get a situation where the parties locally compute and open $\vec{z} = g(\vec{x}, y)$, where g is a composition of linear combinations and truncations (it would be $g(\vec{x}, y) = f(\vec{x}) - y$ in the protocols Π_{verify} and $\Pi_{verify}{}^*$), but now $\vec{z} = [z]$ is a one-element vector, where $z \in \{0, 1\}$ is just a single bit.

Both solutions in turn introduce more alleged zeroes coming either from the multiplication triples (in $\mathbb{Z}_q$) or the trusted bits (in $\mathbb{Z}_{2^m}$). As we have shown in Proposition 5.8, it is safe to open alleged zeroes coming from the tuples.

Complaining about alleged zero. Another problem comes in the case where weak_open of $\vec{z}$ fails. If there is a conflict between the prover P and some verifier V_i, and P is forced in this way to open up to $t-1$ shares issued to corrupted verifiers, then each honest party is able to reconstruct all the commitments from the t shares that they now hold.

We modify $\Pi_{verify}{}^*$ as follows. Instead of committing the value x over $\mathbb{Z}_{2^m}$ directly to $\mathcal{F}_{commit}$, a party first generates a random value $x_1 \xleftarrow{\$} \mathbb{Z}_{2^m}$, computes $x_2 \xleftarrow{\$} x - x_1$, and then commits x_1 and x_2 separately. This is done to all the inputs, randomness, communication, outputs, and also the precomputed tuples for all parties.

Let now g be a linear combination s.t $g(\vec{x}) = \vec{0}$ is the expected alleged zero vector, and $\vec{x}$ is the vector of committed values, each $x = x_1 + x_2$ committed as x_1 and x_2. The vector $g(\vec{x})$ takes into account all the additional checks related to the bit decompositions that turn the initial alleged zeroes to a single bit $z \in \{0, 1\}$. Instead of opening $\vec{z} = g(\vec{x}) = g(\vec{x_1} + \vec{x_2})$, the parties do the following:

1. Open $\vec{z_1} = g(\vec{x_1})$. If weak opening succeeds, there is no additional leakage. If weak opening fails, then the parties attempting to tamper with the shares will be discarded from $\mathcal{P}$. Up to $t-1$ shares of $\vec{x_1}$ will be revealed. Hence it may happen that $\vec{x_1}$ leaks to honest parties, but it is uniformly distributed so far. And if the adversary succeeds in leaking $\vec{x_1}$, then *all* corrupted parties will be discarded from $\mathcal{P}$.

2. Open $\vec{z_2} = g(\vec{x_2})$. If weak opening fails, again, up to $t-1$ shares of $\vec{x_2}$ will be revealed. However, even knowing $\vec{x_2}$, the honest parties are unable to reconstruct $\vec{x} = \vec{x_1} + \vec{x_2}$ unless they know $\vec{x_1}$. As described above, they would be able to get $\vec{x_1}$ only if all $t-1$ parties were discarded from $\mathcal{P}$, and in this case $\vec{x_2}$ would not be leaked to honest parties.

After $\vec{z_1}$ and $\vec{z_2}$ are opened, each party reconstructs $\vec{z} = \vec{z_1} + \vec{z_2}$.

In the next sections, we define the protocols including these modifications in more details, and prove their security.

The Commitment Protocol $\Pi_{commit}{}^*$

We define $\mathcal{F}_{commit}{}^*$ similarly to $\mathcal{F}_{commit}^{weak}{}^*$, with the difference that no information is leaked to honest parties about the initial commitments from which the opened value has been computed. We build $\Pi_{commit}{}^*$ on top of $\mathcal{F}_{commit}^{weak}{}^*$. This protocol is given in Figure 5.32.

Proposition 5.10. Let t be the upper bound on active coalition size. Assuming $t < n/2$, the protocol $\Pi_{commit}{}^*$ t-WCP-realizes $\mathcal{F}_{commit}{}^*$ in $\mathcal{F}_{commit}^{weak}{}^*$-hybrid model.

In ${\Pi_{commit}}^*$, each party works locally with unique identifiers id, encoding the bit size $m(id)$ of the ring in which the value is shared, and the parties $p(id)$ and $p'(id)$ that know the shared value. ${\Pi_{commit}}^*$ works on top of ${\mathcal{F}_{commit}^{weak}}^*$.

• **Initialization:** On input $(\mathsf{init}, \hat{m}, \hat{p}, \hat{p'})$, for each identifier $id \in \mathsf{Dom}(\hat{m}) = \mathsf{Dom}(\hat{p}) = \mathsf{Dom}(\hat{p'})$, define two identifiers id_1 and id_2. Assign $m(id_1) = m(id_2) \leftarrow \hat{m}(id)$, $p(id_1) = p(id_2) \leftarrow \hat{p}(id)$, $p'(id_1) = p'(id_2) \leftarrow \hat{p'}(id)$. Send $(\mathsf{init}, m, p, p')$ to ${\mathcal{F}_{commit}^{weak}}^*$.

• **Commit:** On input (commit, id, x), $P_{p(id)}$ generates random $x_1 \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, computes $x_2 \leftarrow x - x_1$ in $\mathbb{Z}_{2^{m(id)}}$, and sends $(\mathsf{commit}, id_1, x_1)$ and $(\mathsf{commit}, id_2, x_2)$ to ${\mathcal{F}_{commit}^{weak}}^*$.

• **Mutual Commit:** On input $(\mathsf{mcommit}, id, id', x)$, $P_{p(id)}$ generates random $x_1 \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, computes $x_2 \leftarrow x - x_1$ in $\mathbb{Z}_{2^{m(id)}}$, and sends $(\mathsf{mcommit}, id_1, {id'}_1, x_1)$ and $(\mathsf{mcommit}, id_2, {id'}_2, x_2)$ to ${\mathcal{F}_{commit}^{weak}}^*$. On input $(\mathsf{mcommit}, id, id', x')$, after obtaining x_1 and x_2 from ${\mathcal{F}_{commit}^{weak}}^*$, $P_{p'(id)}$ computes $x \leftarrow x_1 + x_2$ in $\mathbb{Z}_{2^{m(id)}}$, and checks $x = x'$, broadcasting (bad, id, id') if $x \neq x'$.

• **Weak Open:** On input $(\mathsf{weak\_open}, id)$, each party sends $(\mathsf{weak\_open}, id_1)$ and $(\mathsf{weak\_open}, id_2)$ to ${\mathcal{F}_{commit}^{weak}}^*$. Each party receives x_1 and x_2 from ${\mathcal{F}_{commit}^{weak}}^*$. If at least one of them is $(id_k, \perp)$, output $(id, \perp)$ to $\mathcal{Z}$. Otherwise, output $x_1 + x_2$ to $\mathcal{Z}$.

• **Open:** On input (open, id):

1. Send (open, id_1) to ${\mathcal{F}_{commit}^{weak}}^*$, receiving back x_1. For all messages $(\mathsf{cheater}, k)$ coming from ${\mathcal{F}_{commit}^{weak}}^*$, assign $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$, and output $(\mathsf{cheater}, k)$ to $\mathcal{Z}$.
2. Send (open, id_2) to ${\mathcal{F}_{commit}^{weak}}^*$, receiving back x_2. For all messages $(\mathsf{cheater}, k)$ coming from ${\mathcal{F}_{commit}^{weak}}^*$, assign $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$, and output $(\mathsf{cheater}, k)$ to $\mathcal{Z}$.
3. Output $x = x_1 + x_2$ in $\mathbb{Z}_{2^{m(id)}}$.

• **Other operations:** For any other input $(task, id)$, all parties send $(task, id_1)$ and $(task, id_2)$ to ${\mathcal{F}_{commit}^{weak}}^*$.

Figure 5.32: Real Protocol ${\Pi_{commit}}^*$

Let $comm$, $pubv$, $deriv$ be the local arrays of $\mathcal{S}_i$. On all inputs, except (open, id), the behaviour of $\mathcal{S}_i$ is defined similarly to ${\mathcal{S}_{commit}^{weak}}^*$.

• **Open:** $\mathcal{S}_i$ needs to simulate sending (open, id_1) and (open, id_2) to ${\mathcal{F}_{commit}^{weak}}^*$. As a side effect, ${\mathcal{F}_{commit}^{weak}}^*$ needs to open all the leaves of $deriv[id]$ to $\mathcal{A}_i^H$ for $\mathcal{C} \neq \mathcal{C}_i$. The particular simulation proceeds as follows:

1. Simulate sending (open, id_1) to ${\mathcal{F}_{commit}^{weak}}^*$. By definition, all the leaves of $deriv[id]$ are of the form $x = x_1 + x_2$, where x_1 and x_2 are distributed uniformly, if only one of them has been seen. $\mathcal{S}_i$ samples all revealed shares from uniform distribution. Any party P_k for which $(\mathsf{cheater}, k)$ comes from ${\mathcal{F}_{commit}^{weak}}^*$ is discarded from the protocol, i.e. $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$, $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$.
2. Send (open, id_2) to ${\mathcal{F}_{commit}^{weak}}^*$, receiving back x_2. $\mathcal{S}_i$ samples all revealed shares from uniform distribution. Any party P_k for which $(\mathsf{cheater}, k)$ comes from ${\mathcal{F}_{commit}^{weak}}^*$ is discarded from the protocol, i.e. $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k\}$, $\mathcal{C} \leftarrow \mathcal{C} \setminus \{k\}$.

• **Other operations:** For any input involving x, $\mathcal{S}_i$ generates $x_1 \xleftarrow{\$} \mathbb{Z}_{2^{m(id)}}$, and computes $x_2 \leftarrow x - x_1$ in $\mathbb{Z}_{2^{m(id)}}$. $\mathcal{S}_i$ simulates the analogous operations of ${\mathcal{F}_{commit}^{weak}}^*$ applied to x_1 and x_2, similarly to ${\mathcal{S}_{commit}^{weak}}^*$.

Figure 5.33: The simulator ${\mathcal{S}_{commit}}^*(i)$

Proof. We use the simulator $\mathcal{S}_i = \mathcal{S}_{commit}{}^*(i)$ described in Figure 5.33. All the operations besides strong opening are simulated analogously to $\mathcal{S}_{commit}^{weak}{}^*$, so we do not give here a proof of their correct simulation. We only discuss the strong opening.

Simulatability. $\mathcal{S}_i$ needs to simulate sending (open, id_1) and (open, id_2) to $\mathcal{F}_{commit}^{cheap}{}^*$. By definition of $\mathcal{F}_{commit}^{cheap}{}^*$, if all $t-1$ corrupted parties reveal themselves during the opening, the commitments from which the value $comm[id_1]$ (or $comm[id_2]$) were computed will be output to $\mathcal{A}_i^H$ such that $i \neq 1$. Hence for $i \neq 1$, the simulator $\mathcal{S}_i$ should generate these values itself.

1. When simulating sending (open, id_1) to $\mathcal{F}_{commit}^{weak}{}^*$, up to $t-1$ messages $(\mathsf{cheater}, k)$ may come from $\mathcal{A}^L$. If $t-1$ such messages have come, the leaves x of $deriv[id]$ are output to $\mathcal{A}_i^H$. For each commitment $x = x_1 + x_2$, there are two possibilities for the value x_1.
   - If x_2 has already been delivered to $\mathcal{A}_i^H$ when opening some other value $comm[id']$ before (such that $deriv[id']$ also contained the value x in one of its leaves), then the messages $(\mathsf{cheater}, k)$ should have already been simulated for at least $t-1$ parties P_k, $k \in \mathcal{C}$, since x_2 would not be delivered to $\mathcal{A}_i^H$ otherwise. Since $|\mathcal{C}| \leq t-1$, it would be impossible for $\mathcal{A}^L$ to force opening x_1 now.
   - If x_2 has not been delivered to $\mathcal{A}_i^H$ yet, then x_1 can be sampled by $\mathcal{S}_i$ from a uniform distribution.
2. When sending (open, id_2) to $\mathcal{F}_{commit}^{weak}{}^*$, again, up to $t-1$ shares of $\mathcal{C}$ may need to be opened to $\mathcal{A}_i^H$. This case is symmetric to (open, id_1), and x_1 is now treated as the value that might have already been opened before.

To summarize, either x_1 or x_2 may be opened to $\mathcal{A}_i^H$, but not both. The value x remains unknown to $\mathcal{A}_i^H$.

Correctness. For each x that $\mathcal{F}_{commit}{}^*$ outputs to P_k for $k \in \mathcal{C}_i$, $\mathcal{S}_i$ has simulated exactly the same x in its interaction with $\mathcal{A}_i^H$. For $k \notin \mathcal{C}_i$, $\mathcal{S}_i$ has not leaked to $\mathcal{A}_i^H$ any information about x, even for $p(id) \in \mathcal{C}$. □

Observation 5.5. $\Pi_{commit}{}^*$ uses $\mathcal{F}_{commit}^{weak}{}^*$ as a subroutine to perform the same operations that $\mathcal{F}_{commit}^{weak}{}^*$ does. In all cases, the corresponding operation of $\mathcal{F}_{commit}^{weak}{}^*$ is called twice. Therefore, compared to covertly secure $\Pi_{commit}^{weak}{}^*$, the multiplicative overhead of actively secure $\Pi_{commit}{}^*$ is 2.

The Verification Protocol $\Pi_{verify}^{strong*}$

We define $\mathcal{F}_{verify}^{strong*}$ similarly to $\mathcal{F}_{verify}{}^*$, and the only difference is that, instead of outputting $f(\vec{x}) - y$ to all the adversaries, it outputs just a single bit denoting

• **Initializing subroutine protocols:** The subroutines $\mathcal{F}_{transmit}{}^*$, $\mathcal{F}_{pre}{}^*$, and $\mathcal{F}_{commit}{}^*$ are initialized as in the case of $\Pi_{verify}{}^*$. In addition, for each expected alleged zero $z \in \mathbb{Z}_{2^m}$, the following precomputed tuples need to be generated by $\mathcal{F}_{pre}{}^*$:

1. m trusted bits shared over $\mathbb{Z}_{2^m}$.
2. m one-bit multiplication triples in $\mathbb{Z}_2$.

• **Verification:** On input (verify, id), the parties act in the same way as for $\Pi_{verify}{}^*$ of Figure 4.31, up to the opening of the final alleged zeroes. Let $id_1^z, \ldots, id_\ell^z$ be the identifiers of obtained alleged zeroes corresponding to the difference $f(\vec{x}) - y$, as used by $\mathcal{F}_{commit}{}^*$ (i.e. excluding the alleged zeroes that check the correctness of broadcasts). For $i \in [\ell]$, let $(z_{i1}, \ldots, z_{im})$ be the bit decomposition of the alleged zero $z_i = comm[id_i^z]$ over $\mathbb{Z}_{2^m}$.

1. In the first round, the prover additionally broadcasts $\hat{\vec{z}}_i = [c_{i1}, \ldots, c_{im}]$ for all $i \in [\ell]$, where $c_{ik} \in \{0, 1\}$ denotes whether $b_{ik} = z_{ik}$ for the trusted bit b_{ik}. It also broadcasts the differences $\hat{a}_{ik} = \prod_{l,j=1}^{i,k}(1 - z_{lj}) - a_{ik}$ and $\hat{b}_{ik} = (1 - z_{i(k+1)}) - b_{ik}$ (take $z_{(i+1)1}$ if $k = m$) in $\mathbb{Z}_2$ for the precomputed multiplication triples (a_{ik}, b_{ik}, c_{ik}) for all $i \in [\ell]$, $k \in [m]$.
2. In the second round, instead of directly sending (open, id_i^z) to $\mathcal{F}_{commit}{}^*$ for all $i \in [\ell]$, the parties use the broadcast bits c_{ik} to do the bit decomposition of z_i, as it is done in Π_{verify} and $\Pi_{verify}{}^*$. Let id_{ik}^z be the identifier such that $z_{ik} = comm[id_{ik}^z]$ in $\mathcal{F}_{commit}{}^*$. Then the parties send $(id_{ik}^z = (1 - id_{ik}^z) \bmod 2$ to $\mathcal{F}_{commit}{}^*$. The parties use the precomputed multiplication triples over $\mathbb{Z}_2$ to find the product of all values $comm[id_{ik}^z]$. Let id^z be the identifier such that $comm[id^z]$ the final answer. Finally, the parties send (open, id_i^z) to $\mathcal{F}_{commit}$ for all alleged zeroes that not related to $f(\vec{x}) - y$. If all values returned by $\mathcal{F}_{commit}{}^*$ are 0, send (open, id^z) to $\mathcal{F}_{commit}{}^*$ (otherwise, output 0 to $\mathcal{Z}$). The final value returned by $\mathcal{F}_{commit}{}^*$ is output to $\mathcal{Z}$.

• **Other operations:** The parties act in the same way as in $\Pi_{verify}{}^*$.

Figure 5.34: The protocol $\Pi_{verify}^{strong*}(i)$

whether $f(\vec{x}) - y = 0$. We provide an updated version $\Pi_{verify}^{strong*}$ of $\Pi_{verify}{}^*$ and prove that it WCP-implements $\mathcal{F}_{verify}^{strong*}$. The protocol is depicted in Figure 5.34.

The protocol $\Pi_{verify}^{strong*}$ uses $\mathcal{F}_{commit}{}^*$ instead of $\mathcal{F}_{commit}^{weak}{}^*$, and its subroutine $\mathcal{F}_{pre}{}^*$ will also commit the generated precomputed tuples to $\mathcal{F}_{commit}{}^*$. The only significant change that we need to do is the final alleged zero check.

Proposition 5.11. Let t be the upper bound on active coalition size. Assuming $t < n/2$, the protocol $\Pi_{verify}^{strong*}$ t-WCP-realizes $\mathcal{F}_{verify}{}^*$ in $\mathcal{F}_{transmit}{}^*$-$\mathcal{F}_{commit}{}^*$-$\mathcal{F}_{pre}{}^*$-hybrid model.

Proof. We use the simulator $\mathcal{S}_i = \mathcal{S}_{verify}^{strong*}(i)$ described in Figure 5.35. It runs a local copy of $\Pi_{verify}^{strong*}$, and local copies of $\mathcal{F}_{transmit}{}^*$, $\mathcal{F}_{commit}{}^*$, $\mathcal{F}_{pre}{}^*$.

Simulatability. The simulation of additional alleged zeroes, related to the bit decompositions and multiplications that replace $\vec{z}$ with a single bit $z \in \{0, 1\}$, is done in exactly the same way as for $\mathcal{S}_{verify}{}^*$, so we do not repeat the proof here. The only additional alleged zero that is opened besides them is the one bit z that

• **Initialization:** On input $(\mathsf{init}, f, \vec{xid}, p, p')$ from $\mathcal{F}_{verify}^{strong*}$, $\mathcal{S}_i$ simulates ${\mathcal{F}_{pre}}^*$ to generate all the necessary tuples, including the additional trusted bit and multiplication triples. If execution of ${\mathcal{F}_{pre}}^*$ has not failed, then $\mathcal{A}_i^H$ expects that all (valid) precomputed tuples for $p(id) \in \mathcal{C}$ are copied to ${\mathcal{F}_{commit}}^*$. If the tuple generation fails, $\mathcal{S}_i$ sends (stop) to $\mathcal{F}_{verify}^{strong*}$.

• **Verification:** $\mathcal{S}_i$ decomposes $f(id)$ to basic operations $f_1, \ldots, f_N$, and defines the additional identifiers $id_i^{\mathsf{x}k}$, $id_i^{\mathsf{y}k}$, $id_i^{\mathsf{z}k}$ as the honest parties do.

1. For $p(id) \in \mathcal{C}_i$, $\mathcal{S}_i$ computes all the intermediate values $comm[id_i^{\mathsf{x}k}]$ and $comm[id_i^{\mathsf{y}k}]$. It uses them to compute $\hat{\vec{x}}$, including the additional values of $\Pi_{commit}^{strong*}$. The broadcast of $p(id) \notin \mathcal{C}_i$ is simulated by $\mathcal{S}_i$ exactly in the same way as it was done by ${\mathcal{S}_{verify}}^*$, sampling each entry of $\hat{\vec{x}}$ from uniform distribution over corresponding ring, including the additional values of $\Pi_{commit}^{strong*}$. For $p(id) \in \mathcal{C}$, the values $\hat{\vec{x}}^*$ are chosen by $\mathcal{A}^L$. $\mathcal{S}_i$ simulates broadcasting obtained $\hat{\vec{x}}^*$ and $\hat{\vec{x}}$ through $\mathcal{F}_{transmit}$ similarly to $\mathcal{S}_{verify}$.
2. The local computation of the verifiers depends on f_i, and $\mathcal{S}_i$ just simulates using ${\mathcal{F}_{commit}}^*$ to compute certain local operations. After the computation of $\vec{z}$ has finished, instead of simulating its opening, $\mathcal{S}_i$ continues with the simulation of local computation that is extended by ${\Pi_{verify}}^*$. The simulation of these additional steps is analogous to the simulation of locally computing f_i, since they do not require interaction with $\mathcal{A}_i^H$ and $\mathcal{A}^L$. In the end, $\mathcal{S}_i$ simulates to $\mathcal{A}_i^H$ the opening of all the additional alleged zeroes that come from the broadcasts. They are chosen similarly to ${\mathcal{F}_{verify}}^*$, i.e. $z = 0$ if $p(id) \notin \mathcal{C}$, and $z = \hat{x}^* - \hat{x}$ or $z = \sum_{k=1}^{m}((c_k^* - c_k) \bmod 2)$ for the broadcast $\hat{x}$ and c_k corresponding to these new tuples if $p(id) \in \mathcal{C}$. If all opened values are 0, it simulates to $\mathcal{A}_i^H$ the opening of the bit b that was obtained from $\mathcal{F}_{verify}^{strong*}$.

• **Other operations:** $\mathcal{S}_i$ acts in the same way as ${\mathcal{S}_{verify}}^*$ does.

Figure 5.35: The simulator $\mathcal{S}_{verify}^{strong*}$

is supposed to verify whether $f(\vec{x}) - y = 0$. It suffices to prove that $f(\vec{x}) - y = 0$ iff $z = 0$ and all the broadcast values have passed the first verification.

1. Assuming that the broadcasts related to trusted bits were correct, after the parties have computed the bit decomposition of z_i using these trusted bits, the bits z_{ik} of each alleged zero are stored in ${\mathcal{F}_{commit}}^*$ as $comm[id_{ik}^{\mathsf{z}}]$.
2. Assuming that the broadcasts corresponding to multiplication triples were correct, the product $\prod_{i,k}^{\ell,m}(1 - z_{ik})$ is stored in ${\mathcal{F}_{commit}}^*$ as $comm[id^{\mathsf{z}}]$. Hence it should be $z = \prod_{i,k}^{\ell,m}(1 - z_{ik})$. This product equals 0 iff $z_{ik} = 0$ for all i, k, and this is in turn equivalent to the statement $\vec{z} = \vec{0}$, where $\vec{z}$ has been computed in the same way as in Π_{verify} and ${\Pi_{verify}}^*$, so $\vec{z} = [f(\vec{x}) - y]$.

Correctness. Similarly to $\mathcal{F}_{verify}$ of Figure 4.30, the inputs / messages of $p(id) \notin \mathcal{C}$, the randomness chosen by $\mathcal{F}_{verify}$, and the inputs / messages of $p(id) \in \mathcal{C}$ chosen by $\mathcal{A}$ are all stored in ${\mathcal{F}_{commit}}^*$. In addition, the precomputed tuples are also stored in the same ${\mathcal{F}_{commit}}^*$ by definition of ${\mathcal{F}_{pre}}^*$. ${\mathcal{F}_{commit}}^*$ may

now be used as a black box, doing computation on all these commitments. It remains to prove that, if all these values are committed properly, then $\Pi_{verify}^{strong*}$ does verify the computation of $f(id)$ on input (verify, id).

For all parties following the protocol, including non-cheating corrupted parties, $\mathcal{S}_i$ takes $z = 0$ for all alleged zeroes, so the verification definitely passes for these parties. We show that the converse also holds. Suppose that the final check passes, i.e. $\vec{z} = \vec{0}$.

- For all broadcast-related alleged zeroes, $z = 0$ implies that the broadcast of the first round was correct. If $p(id) \in \mathcal{C}$, $\mathcal{S}_i$ has chosen $z = x'^* - x'$ and $z = \sum_{k=1}^{m} 2^{k-1}((c_k^* - c_k) \bmod 2)$. In this way, $z = 0$ implies $x^* = x'$ and $c_k^* = c_k$ for all k, meaning that $\mathcal{A}^L$ has not cheated with the broadcasts of $P_{p(id)}$.
- We have shown before, that if all broadcast-related alleged zeroes are 0, and $z = 0$ for the last alleged zero z, then $f(\vec{x}) - y = 0$.

These two arguments reduce the proof to the correctness of Proposition 5.8. □

Proposition 5.12. Let ℓ be the length of the alleged zero vector, and 2^m the size of the largest ring from which the alleged zeroes come. Compared to the covertly secure protocol ${\Pi_{verify}}^*$, the actively secure $\Pi_{verify}^{strong*}$ has the following overheads:

- *Offline:* additive $\mathsf{prc}_{\mathsf{bit},m}^{\ell m} \otimes \mathsf{prc}_{\mathsf{triple},m}^{\ell m}$, followed by multiplicative 2;
- *Online:* multiplicative overhead 5.

Proof. We count the total number of additional operations of $\Pi_{verify}^{strong*}$:

- *Offline:* The total number of additional tuples is taken directly from Figure 5.34. There are ℓm trusted bits and ℓm one-bit multiplication triples, so the cost of their generation is $\mathsf{prc}_{\mathsf{bit},m}^{\ell m} \otimes \mathsf{prc}_{\mathsf{triple},m}^{\ell m}$. This cost is doubled due to the double expense of commitments using ${\mathcal{F}_{commit}}^*$.
- *Online:* During the broadcasting of hints, ℓm bits need to be broadcast by the prover for the additional trusted bits, and $2\ell m$ bits for the multiplication triples. The additional complexity of these broadcasts is $\mathsf{bc}_{3\ell m}$.

  During the final check, the old alleged zero vector $\vec{z}$ is substituted by a single bit, but there are now two alleged zero bits coming from the additional multiplication, and one bit coming from the bit decomposition, for each of the ℓm bits of $\vec{z}$. The total cost is $\mathsf{bc}_{3\ell \cdot \mathsf{sh}_n \cdot m}^{\otimes n}$. Since the old alleged zero vector of ℓm bits no longer needs to be opened, the additive overhead is $\mathsf{bc}_{2\ell \cdot \mathsf{sh}_n \cdot m}^{\otimes n}$, and the multiplicative overhead is 3. Due to the double expense of opening using ${\mathcal{F}_{commit}}^*$, the total multiplicative overhead is 6.

Since the number of bits in the alleged zero vector is at least the number of bits in the hint vector (see Table 4.7), the multiplicative overhead of the hint broadcasts is upper bounded by 4, and the total verification overhead by 5.

The multiplicative overhead of all commitments is $2 < 5$. □

Proposition 5.13. Let M_c be the total number of bits sent in the execution phase of the original passively secure protocol. Compared to the UC protocol Π_{verify} built on top of Π_{commit}, Π_{rnd} and Π_{pre}, the protocol $\Pi_{verify}^{strong*}$ built on top of $\Pi_{transmit}{}^*$, $\Pi_{commit}{}^*$ and $\Pi_{pre}{}^*$ has the following costs:

- *Preprocessing:* $(X_{pre}^{\otimes 6n} \otimes \mathsf{prc}_{\mathsf{bit},m}^{2\ell m} \otimes \mathsf{prc}_{\mathsf{triple},m}^{2\ell m}) \otimes (\mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 2n(t+1)} \oplus \mathsf{fwd}_{\mathsf{sh}_n \cdot M_c}^{\otimes 3n(t+1)} \oplus \mathsf{tr}_{\mathsf{sh}_n \cdot M_c}^{\otimes 3n(t+1)})$, where X_{pre} is the cost of the preprocessing of Π_{verify}.
- *Execution:* No overheads in the cheap mode. $X_{exec}^{\otimes n}$ in the expensive mode, where X_{exec} is the cost of the expensive mode of the execution of Π_{verify}.
- *Postprocessing:* $X_{post}^{\otimes 5n}$, where X_{post} is the cost of either the cheap or the expensive mode of the postprocessing of Π_{verify}.

Proof. The costs of covertly secure $\Pi_{verify}{}^*$ are taken from Proposition 5.9. They are extended with the overheads of actively secure protocols, taken from Proposition 5.12. The preprocessing overhead is taken from its offline part, and the postprocessing overhead is taken from its online part. The execution phase has no overhead compared to covert security since the same $\mathcal{F}_{transmit}{}^*$ is used. □

The Verification Protocol $\Pi_{active}{}^*$

We define an ideal functionality for actively secure computation. We have the same settings as in $\mathcal{F}_{vmpc}$ of Figure 4.1. The circuit C_{ij}^{ℓ} computes the ℓ-th round messages $\vec{m}_{ij}^{\ell}$ to the party $j \in [n]$ from the input $\vec{x}_i$, randomness $\vec{r}_i$ and the messages $\vec{m}_{j'i}^{k}$ ($k < \ell$) that P_i has received before. All values $\vec{x}_i, \vec{r}_i, \vec{m}_{ij}^{\ell}$ are vectors over rings $\mathbb{Z}_N$. The messages received during the r-th round comprise the output of the protocol. Let $\mathcal{C}$ be the set of corrupted parties, and let $\mathcal{H} := [n] \setminus \mathcal{C}$. The ideal functionality $\mathcal{F}_{active}{}^*$, running in parallel with the environment $\mathcal{Z}$ (specifying the computations of all parties in the form of circuits and the inputs of honest parties), as well as the adversary $\mathcal{A}_S = (\mathcal{A}_S{}_1^H, \ldots, \mathcal{A}_S{}_n^H, \mathcal{A}_S{}^H, \mathcal{A}_S{}^L)$, is given in Figure 5.36. Note that we allow leaking to honest parties the values $\vec{m}_{ij}^{\ell}$ that they would have received anyway if the adversary was passive.

The protocol $\Pi_{active}{}^*$ implementing actively secure computation is given in Figure 5.37. The only difference from Π_{vmpc} of Figure 4.35 is that the outgoing messages of the parties are verified on each round, and that $\mathcal{F}_{verify}^{strong*}$ is used as a subroutine instead of $\mathcal{F}_{verify}$.

• **In the beginning**, $\mathcal{F}_{active}{}^*$ gets from $\mathcal{Z}$ for each party P_i the message $(\mathsf{circuits}, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ and forwards it to all adversaries ${\mathcal{A}_S}_j^H$.

For each $i \in [n]$, $\mathcal{F}_{active}{}^*$ randomly generates $\vec{r}_i$. For all $i \in [n]$, it sends $(\mathsf{randomness}, i, \vec{r}_i)$ to ${\mathcal{A}_S}_{c(i)}^H$. At this point, ${\mathcal{A}_S}^L$ **may stop** the functionality. If it continues, then for each $i \in \mathcal{H}$ [resp $i \in \mathcal{C}$], $\mathcal{F}_{active}{}^*$ gets $(\mathsf{input}, \vec{x}_i)$ from $\mathcal{Z}$ [resp. ${\mathcal{A}_S}^L$]. The messages $(\mathsf{input}, \vec{x}_i)$ are delivered to the adversary $\mathcal{A}_{c(i)}^H$.

• **For each round** $\ell \in [r]$, $i, j \in [n]$, $\mathcal{F}_{active}{}^*$ uses C_{ij}^{ℓ} to compute the message $\vec{m}_{ij}^{\ell}$. For all $i \in [n]$, $j \in [n]$, it sends $\vec{m}_{ij}^{\ell}$ to $\mathcal{A}_{c(j)}^H$. For each $j \in \mathcal{C}$ and $i \in \mathcal{H}$, it receives $\vec{m}_{ji}^{*\ell}$ from ${\mathcal{A}_S}^L$. If $\vec{m}_{ji}^{*\ell} = \top$, it takes $\vec{m}_{ji}^{*\ell} = \vec{m}_{ji}^{\ell}$.

If $\vec{m}_{ij}^{\ell} \neq \vec{m}_{ij}^{*\ell}$ for at least one message, $\mathcal{F}_{active}{}^*$ defines $\mathcal{M}' = \{i \in \mathcal{C} \mid \exists j \in [n] : \vec{m}_{ij}^{\ell} \neq \vec{m}_{ij}^{*\ell}\}$. In this case the outputs are not sent to $\mathcal{Z}$. $\mathcal{F}_{active}{}^*$ outputs $(\mathsf{cheaters}, \mathcal{M}')$ to each adversary $\mathcal{A}_k^H$.

• **After** r **rounds**, $\mathcal{F}_{active}{}^*$ sends $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$ to each party P_i with $i \in \mathcal{H}$. In this case $\mathcal{F}_{active}{}^*$ takes $\mathcal{B}_0 = \emptyset$.

Alternatively, **at any time** before outputs are delivered to parties, ${\mathcal{A}_S}^L$ may send $(\mathsf{stop}, \mathcal{B}_0)$ to $\mathcal{F}_{active}{}^*$, with $\mathcal{B}_0 \subseteq \mathcal{C}$. In this case the outputs are not sent.

• **Finally**, for each $i \in \mathcal{H}$, ${\mathcal{A}_S}^L$ sends $(\mathsf{blame}, i, \mathcal{B}_i)$ to $\mathcal{F}_{active}{}^*$, with $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$, where $\mathcal{M} = \mathcal{B}_0 \cup \mathcal{M}'$. $\mathcal{F}_{active}{}^*$ forwards this message to P_i.

Figure 5.36: The ideal functionality $\mathcal{F}_{active}{}^*$ for verifiable computations

Proposition 5.14. Let t be the upper bound on active coalition size. Assuming $t < n/2$, the protocol $\Pi_{active}{}^*$ t-WCP-realizes $\mathcal{F}_{active}{}^*$ in $\mathcal{F}_{verify}^{strong*}$-hybrid model.

Proof. We use the simulator $\mathcal{S}_i = \mathcal{S}_{active}{}^*(i)$ described in Figure 5.38. The simulator runs a local copy of $\Pi_{active}{}^*$, together with a local copy of $\mathcal{F}_{verify}^{strong*}$.

Simulatability. The commitment of inputs, messages, and randomness is reduced to $\mathcal{F}_{verify}^{strong*}$. For $\mathcal{S}_k$, it is only important to know the committed values for $i \in \mathcal{C}_k$ (and also $\vec{m}_{ij}^{\ell}$ for $j \in \mathcal{C}_k$). All these messages are delivered to $\mathcal{S}_k$ by $\mathcal{F}_{active}{}^*$. If $i \in \mathcal{C}$, then $\mathcal{S}_k$ additionally gets $\vec{m}_{ij}^{*\ell}$ from $\mathcal{A}^L$, and outputs to $\mathcal{A}_k^H$ the value $\vec{m}_{ij}^{*\ell}$ instead of $\vec{m}_{ij}^{\ell}$. Since we allowed $\mathcal{F}_{verify}^{strong*}$ to output the message $\vec{m}_{ij}^{\ell}$ to ${\mathcal{A}_S}_{c(j)}^H$ before ${\mathcal{A}_S}^L$ has made its choice of $\vec{m}_{ij}^{*\ell}$, the simulator is able to simulate $\vec{m}_{ij}^{*\ell}$ to $\mathcal{A}_{c(j)}^H$ in the case $\vec{m}_{ij}^{*\ell} = \top$ by taking $\vec{m}_{ij}^{*\ell} = \vec{m}_{ij}^{\ell}$.

On each round, after all necessary commitments are made, $\mathcal{S}_k$ simulates the side-effect of $\mathcal{F}_{verify}^{strong*}$ that outputs the bit denoting $f(\vec{x}) = y$ for the committed inputs $\vec{x}$ and the committed output y. For $i \notin \mathcal{C}$, these values are 0. For $i \in \mathcal{C}$, if $(\mathsf{cheaters}, \mathcal{M}')$ comes from $\mathcal{F}_{active}{}^*$, $\mathcal{S}_i$ takes $b = 0$ iff $i \in \mathcal{M}'$. In this way, the party P_i cheats in the ideal world iff $\mathcal{F}_{verify}^{strong*}$ outputs 0 in the real world. If any party P_k causes $\mathcal{F}_{verify}^{strong*}$ or $\mathcal{F}_{transmit}{}^*$ to output $(\mathsf{cheater}, k)$, then $\mathcal{S}_i$ does not need to simulate the verification of computation of P_k.

All verifiable functions f of $\mathcal{F}_{verify}^{strong*}$ correspond to the computation of some output of a circuit C_{ij}^{ℓ} w.r.t. the committed inputs, randomness, and messages.

In ${\Pi_{active}}^*$, each party P_i maintains a local array mlc_i of length n, into which it marks the parties that have been detected in violating the protocol rules. Initially, $mlc_i[k] = 0$ for all $k \in [n]$. If P_k has been detected in cheating, P_i writes $mlc_i[k] = 1$. ${\Pi_{active}}^*$ uses ${\mathcal{F}_{verify}^{strong}}^*$ as a subroutine.

• **In the beginning**, Each party P_i gets the message $(\mathsf{circuits}, (C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r})$ from $\mathcal{Z}$.

1. *Initializing ${\mathcal{F}_{verify}^{strong}}^*$:* Let the n_{ij}^{ℓ} output wires of the circuit C_{ij}^{ℓ} be enumerated. For all $k \in [n_{ij}^{\ell}]$, the value $id \leftarrow (i, j, \ell, k)$ serves as an identifier for ${\mathcal{F}_{verify}^{strong}}^*$. In addition, for each party P_i, there are identifiers (i, x, k) and (i, r, k) for the enumerated inputs and randomness respectively.

   - For each input wire $id \leftarrow (i, \mathsf{x}, k)$ or $id \leftarrow (i, \mathsf{r}, k)$, let 2^m be the size of the ring in which the wire is defined. Define $f(id) \leftarrow \mathsf{id}_{\mathbb{Z}_{2^m}}$, $\vec{xid}(id) \leftarrow [id]$, $p(id) = p'(id) = i$.
   - For each output wire $id \leftarrow (i, j, \ell, k)$, define $f(id)$ as a function consisting of basic circuit operations (Section 4.2), computing the k-th coordinate of $\vec{m}_{ij}^{\ell} \leftarrow C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^1, \ldots, \vec{m}_{ni}^{\ell-1})$ (this is always possible since every gate of C_{ij}^{ℓ} is by definition some basic operation), $\vec{xid}(id)$ the vector of all the identifiers of $\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^1, \ldots, \vec{m}_{ni}^{\ell-1}$ that are actually used by C_{ij}^{ℓ}, $p(id) = i$, $p'(id) = j$.

   Each party sends $(\mathsf{init}, f, \vec{xid}, p, p')$ to ${\mathcal{F}_{verify}^{strong}}^*$.
2. *Randomness generation:* For each randomness input wire $id \leftarrow (i, \mathsf{r}, k)$, each party sends $(\mathsf{commit\_rnd}, id)$ to ${\mathcal{F}_{verify}^{strong}}^*$.
3. *Input commitment:* For each input wire $id \leftarrow (i, \mathsf{x}, k)$, P_i sends $(\mathsf{commit\_input}, id, \vec{x}_i)$ to ${\mathcal{F}_{verify}^{strong}}^*$, and each other party sends $(\mathsf{commit\_input}, id)$ to ${\mathcal{F}_{verify}^{strong}}^*$.

• **For each round** $\ell \in [r]$, P_i computes $\vec{m}_{ij}^{\ell} = C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^1, \ldots, \vec{m}_{ni}^{\ell-1})$ for all $j \in [n]$, and sends $(\mathsf{send\_msg}, (i, j, \ell, k), m_{ijk}^{\ell})$ to ${\mathcal{F}_{verify}^{strong}}^*$ for all $k \in [|\vec{m}_{ij}^{\ell}|]$. Each other party sends $(\mathsf{send\_msg}, (i, j, \ell, k))$ to ${\mathcal{F}_{verify}^{strong}}^*$. Immediately after that, each party sends $(\mathsf{commit\_msg}, id)$ to ${\mathcal{F}_{verify}^{strong}}^*$.

Alternatively, if a message $(\mathsf{cheater}, k)$ comes from ${\mathcal{F}_{verify}^{strong}}^*$, each party P_i writes $mlc_i[k] \leftarrow 1$. In this case the verification is not run for P_k. The protocol stops after this round (unless the protocol allows to proceed even after some parties are discarded from the execution), and the protocol outputs are not sent to $\mathcal{Z}$. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.

• **After all parties have been committed to their outputs**, the verification starts. For each output wire identifier $id \leftarrow (i, j, \ell, k)$, each party sends (verify, id) to ${\mathcal{F}_{verify}^{strong}}^*$, getting a single bit b from ${\mathcal{F}_{verify}^{strong}}^*$. If $b = 1$, each party writes $mlc_i[k] \leftarrow 0$. Otherwise, it writes $mlc_i[k] \leftarrow 1$, and the protocol does not proceed further (unless the protocol allows to proceed even after some parties are discarded from the execution), and the protocol outputs are not sent to $\mathcal{Z}$. Let $r' \in \{0, \ldots, r-1\}$ be the last completed round.

• **Finally**, each party P_i outputs to $\mathcal{Z}$ the set of parties $\mathcal{B}_i$ such that $mlc_i[k] = 1$ iff $k \in \mathcal{B}_i$. If $r = r'$, it also outputs $(\mathsf{output}, \vec{m}_{1i}^r, \ldots, \vec{m}_{ni}^r)$ to $\mathcal{Z}$.

Figure 5.37: The protocol ${\Pi_{active}}^*$ for actively secure computations

• **In the beginning**, $\mathcal{S}_k$ gets all the circuits $(C_{ij}^{\ell})_{i,j,\ell=1,1,1}^{n,n,r}$ from ${\mathcal{F}_{active}}^*$. These are the same circuits that the parties would have obtained from $\mathcal{Z}$ in Π_{active}.

1. *Initializing* ${\mathcal{F}_{verify}^{strong}}^*$*:* $\mathcal{S}_k$ simulates the initialization of ${\mathcal{F}_{verify}^{strong}}^*$.
2. *Randomness generation:* $\mathcal{S}_k$ simulates sending the messages $(\mathsf{commit\_rnd}, id)$ to ${\mathcal{F}_{verify}^{strong}}^*$ for each input wire $id \leftarrow (i, \mathsf{r}, k)$. For all $i \in [n]$, the randomness $\vec{r}_i$ provided by ${\mathcal{F}_{verify}^{strong}}^*$ is the same as the randomness $\vec{r}_i$ generated by ${\mathcal{F}_{active}}^*$.
3. *Input commitment:* For each input wire $id \leftarrow (i, \mathsf{x}, k)$, $\mathcal{S}_k$ simulates sending $(\mathsf{commit\_input}, id, \vec{x}_i)$ and $(\mathsf{commit\_input}, id)$ to ${\mathcal{F}_{verify}^{strong}}^*$, which is possible without knowing $\vec{x}_i$. For $i \in \mathcal{C}$, the value $\vec{x}_i^*$ is chosen by $\mathcal{A}^L$. $\mathcal{S}_k$ delivers this $\vec{x}_i^*$ to ${\mathcal{F}_{active}}^*$.

• **For each round** $\ell \in [r]$, $\mathcal{S}_k$ needs to simulate parties committing to the messages $\vec{m}_{ij}^{\ell} = C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell-1})$, which should be output to $\mathcal{A}_k^H$ for $j \in \mathcal{C}_k$. For $j \in \mathcal{C}_k$, the message $\vec{m}_{ij}^{\ell}$ comes from ${\mathcal{F}_{active}}^*$. If $i \in \mathcal{C}$, then $\mathcal{S}_k$ gets $\vec{m}_{ij}^{*\ell}$ from $\mathcal{A}^L$ (it takes $\vec{m}_{ij}^{*\ell} = \vec{m}_{ij}^{\ell}$ if $\vec{m}_{ij}^{*\ell} = \top$), and outputs $\vec{m}_{ij}^{*\ell}$ to $\mathcal{A}_k^H$. For $j \notin \mathcal{C}_k$, the commitments are easy to simulate without knowing $\vec{m}_{ij}^{\ell}$.

Alternatively, if a message $(\mathsf{cheater}, k)$ comes from ${\mathcal{F}_{verify}^{strong}}^*$, $\mathcal{S}_k$ writes $mlc_i[k] \leftarrow 1$ for each honest party P_i. In this case the outputs do not have to be sent to $\mathcal{Z}$. $\mathcal{S}_k$ defines $\mathcal{B} = \{k \,|(\mathsf{cheater}, k)$ has been output$\}$, and sends $(\mathsf{stop}, \mathcal{B})$ to ${\mathcal{F}_{active}}^*$ to prevent it from continuing the execution. ${\mathcal{F}_{active}}^*$ outputs $\mathcal{B}$ to each party P_i.

• **After all parties have been committed to their outputs**, for each output wire $id \leftarrow (i, j, \ell, k)$, $\mathcal{S}_k$ simulates sending (verify, id) to ${\mathcal{F}_{verify}^{strong}}^*$. $\mathcal{S}_k$ needs to simulate the output bit b of ${\mathcal{F}_{verify}^{strong}}^*$. $\mathcal{S}_k$ takes $b = 0$ iff a message $(\mathsf{cheaters}, \mathcal{M}')$ has come from ${\mathcal{F}_{active}}^*$, such that $i \in \mathcal{M}'$.

If $k \in \mathcal{C}$, and $b_k = 0$ was simulated as the output of ${\mathcal{F}_{verify}^{strong}}^*$, then $\mathcal{S}_k$ writes $mlc_i[k] \leftarrow 1$ for each honest party P_i, and the simulation stops. Otherwise, it writes $mlc_i[k] \leftarrow 0$. For all $k \notin \mathcal{C}$, it writes $mlc_i[k] \leftarrow 0$.

• **Finally**, if $r = r'$, then each (honest) party P_i should output $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$ to $\mathcal{Z}$. This does not need to be simulated, and we only need to prove the correctness of such outputs.

Figure 5.38: The simulator ${\mathcal{S}_{active}}^*(i)$ for actively secure computations

By definition of ${\mathcal{F}_{verify}^{strong}}^*$, unless at least one message $(\mathsf{cheater}, p(id))$ has been output to each honest party (in this case $p(id) \in \mathcal{C}$), all these values are indeed committed as chosen by the party committing to them.

Since each honest party has followed the protocol and computed C_{ij}^{ℓ} properly, and all its commitments are valid, the differences $f(\vec{x}) - y$ should be 0 for honest parties, and so are easy to simulate.

Correctness. We prove that ${\mathcal{F}_{active}}^*$ outputs exactly the same values as the parties in ${\Pi_{active}}^*$ would. By definition of ${\mathcal{F}_{verify}^{strong}}^*$, there are two kinds of outputs:

1. *The computation output* $(\mathsf{output}, \vec{m}_{1i}^{r}, \ldots, \vec{m}_{ni}^{r})$. Let ℓ be any round. We prove by induction that each message $\vec{m}_{ij}^{\ell}$ seen by the adversary is consistent with the internal state of ${\mathcal{F}_{active}}^*$.

    - *Base:* Initially, there are the inputs $\vec{x}_i$ and the randomness $\vec{r}_i$ in the internal state of ${\mathcal{F}_{active}}^*$. The same values are committed into ${\mathcal{F}_{verify}^{strong}}^*$

in the real protocol. The state of ${\mathcal{F}_{active}}^*$ is consistent with $\mathcal{A}_k^H$'s view of Π_{active}.

- *Step:* By induction hypothesis, the messages $\vec{m}_{ji}^{\ell}$ and the inputs/randomness $\vec{x}_i$,$\vec{r}_i$ of the inner state of ${\mathcal{F}_{active}}^*$ are consistent with the view of $\mathcal{A}_k^H$ of ${\Pi_{active}}^*$. In ${\Pi_{active}}^*$, $\mathcal{A}_k^H$ expects that an honest P_i will now compute each message $\vec{m}^{\ell+1} = C_{ij}^{\ell}(\vec{x}_i, \vec{r}_i, \vec{m}_{1i}^{1}, \ldots, \vec{m}_{ni}^{\ell})$. In the inner state of $\mathcal{F}_{verify}^{strong*}$, the value $\vec{m}^{\ell+1}$ is computed in exactly the same way. If the verification fails, then both ${\mathcal{F}_{active}}^*$ and ${\Pi_{active}}^*$ do not output to $\mathcal{Z}$ anything except the set of blamed parties.

2. *The sets $\mathcal{B}_i$ of blamed parties.* ${\mathcal{F}_{active}}^*$ constructs the set $\mathcal{M}'$ of parties j for whom $\vec{m}_{ij}^{*\ell} \neq \vec{m}_{ij}^{\ell}$ were provided by $\mathcal{S}_k$ on the last round. After that, it receives a couple of messages $(\mathsf{blame}, i, \mathcal{B}_i)$ from $\mathcal{S}_k$, where $\mathcal{B}_i = \mathcal{B}_0 \cup \mathcal{B}_i'$, and $\mathcal{B}_0 = \{k \mid (\mathsf{cheater}, k)$ has come from $\mathcal{F}_{verify}^{strong*}$ during execution $\}$. ${\mathcal{F}_{active}}^*$ expects $\mathcal{M} \subseteq \mathcal{B}_i \subseteq \mathcal{C}$ for $\mathcal{M} = \mathcal{B}_0 \cup \mathcal{M}'$. After this point, the proof becomes analogous to the proof of Lemma 4.11. □

5.5 Summary

We have defined a general framework for representing cryptographic protocols and analyzing their security, that can be viewed as an alternative version of the UC framework. Our model, called WCP, allows to analyze whether the protocol is protected against leaking information of one honest party to another honest party. It helps to avoid some attacks that are not covered neither by the standard UC framework, nor the similar multiple adversary frameworks CP and LUC. The WCP model assumes not the unconditional honesty of uncorrupted parties, but rather their non-collusion with corrupted parties, which is a more realistic assumption. The security definitions are stronger than the standard UC security definitions.

We have proposed some schemes transforming passively secure protocols with one monolithic adversary into actively secure protocols with semihonest majority and multiple adversaries. While the CP and the LUC models require to eliminate any additional subliminal channels that enable the protocol to amplify existing side-channels more than the ideal functionality is able to amplify, in the WCP model it is sufficient to eliminate the situations where protocol specification requires data to be leaked to some honest party.

Although our proposed protocols are insecure in the CP and LUC models since the WCP model is strictly weaker, we think that also CP and LUC would benefit from making some assumptions about the behaviour of (semi)honest parties.

CHAPTER 6

OPTIMIZATION OF SMC PROGRAMS WITH PRIVATE CONDITIONALS

6.1 Chapter Overview

Secure computation platforms are often provided with a programming language that allows a developer to write privacy-preserving applications without having to think on the underlying cryptographic protocols (see Section 2.2.4). The control flow of such programs is expensive to hide, and the attacker in particular knows which choice is being made in conditional statements. Therefore, branching on private values is not straightforward. Instead of choosing only a single branch, all the branches need to be executed in order to conceal the choice. The resulting values of all program variables are chosen from the outcomes of all branches *obliviously* [106, 88] (i.e without leaking information about which choice has been made). Execution of all branches introduces excessive computation, the results of which are actually never used.

If different branches contain identical private operations, then it is reasonable to compute such operations only once. A simple optimization idea, which has not received much attention so far except for [55] in different settings, is to locate identical operations in different branches and try to fuse them into one. Namely, instead of computing the same operation several times and choosing the result obliviously, we can first choose the *inputs* of that operation obliviously, and compute the operation itself only once. The optimization seems very simple, but it is not trivial, since putting some gates together makes it impossible to put some other gates together. Finding the optimal solution is a combinatorial optimization task. In this thesis, we base our optimization on *mixed integer linear programming*, but some greedy heuristics are proposed as well for better performance.

In contrast to the protocol level optimizations that we described in Chapter 4, the methods of this chapter target the program level of secure computation. On this

higher level, cryptographic protocols are used as black boxes, and the precise work of the parties behind the protocol is invisible. Our optimization is very generic and can be applied on the program level without decomposing blackbox operations to arithmetic or boolean circuits.

In this chapter, we consider a simple imperative language with variables typed *public* and *private*. It is allowed to use expressions typed *private* in the conditional statements. We translate a program written in this language into a computational circuit and optimize it, trying to fuse together operations, where the outcome of at most one of them is used in any concrete execution. We apply the optimization to some simple programs that use branching on private variables, and evaluate them on top of the Sharemind SMC platform [17], showing that the optimization is indeed useful in practice.

The protocol that we described in Chapter 4 consists of the preprocessing, the execution, and the verification phases. In the first place, our optimization is intended to improve the execution phase. In particular, it minimizes the total number of bits communicated between the parties. However, this particular optimization reduces not only the communication complexity, but also the local computation, and hence the overheads of the preprocessing and the verification phases. In Section 6.7, we discuss how the same techniques can be applied on the protocol level to reduce the number of verifiable operations directly. Since there are no cases that could be optimized in Sharemind protocols, the discussion part has just a theoretical contribution, and we have not used it for benchmarking.

6.2 Programming Language for SMC

We start from a simple imperative language, given in Figure 6.1, which is just a list of assignments and conditional statements. The variables x in the language are typed either as public or private, these types also flow to expressions. Namely, the expression $f(e_1, \ldots, e_n)$ is private iff at least one of e_i is private. The declassification operation turns a private expression to a public one. An assignment of a private expression to a public variable is not allowed. Only private variables may be assigned inside the branches of private conditions [106, 88]. The syntax c denotes compile-time constants.

During the execution of a program on top of a secret sharing based SMC platform, public values are known to all computation parties, while private values are secret-shared among them [15]. An *arithmetic blackbox function* is an arithmetic, relational, boolean or some other other operation, for which we have implementations for all partitionings of its arguments into public and private values. For example, for integer multiplication, we have the multiplication of public values, and also protocols to multiply two private values, as well as a public and a private

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{stmt} \\
f & ::= & \mathit{arithmetic\ blackbox\ function} \\
\mathit{exp} & ::= & x^{\mathrm{pub}} \mid x^{\mathrm{priv}} \mid c \mid f\ \texttt{(}\ \mathit{exp}^*\ \texttt{)} \mid \texttt{declassify (}\ \mathit{exp}\ \texttt{)} \\
\mathit{stmt} & ::= & x\ \texttt{:=}\ \mathit{exp} \\
& \mid & \texttt{skip} \\
& \mid & \mathit{stmt}\ \texttt{;}\ \mathit{stmt} \\
& \mid & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}
\end{array}
$$

Figure 6.1: Syntax of the imperative language

value [17]. Different kinds of multiplication are represented by different protocols.

The programs in the language of Figure 6.1 cannot all be executed due to the existence of private conditionals. They can be executed after translating them into computational circuits. These circuits are not convenient for expressing looping constructs. Also, our optimization so far does not handle loops. For this reason, we have excluded them from the language. We note that loops with public conditions could in principle be handled inside private conditionals [106].

Let Var be the set of program variables, and Val the set of values that the variables may take. Let $\mathit{State} : \mathit{Var} \to \mathit{Val}$ be a *program state*, which assigns a value to each program variable. The semantics $[\![\cdot]\!]$ defines how executing a program statement modifies the state (while the same notation $[\![\cdot]\!]$ has been used to denote sharing so far, in this chapter we will only use it to denote semantics). Let P be a program written in a language whose syntax is given in Figure 6.1. We define $[\![P]\!] : \mathit{State} \to \mathit{State}$ as follows:

- $[\![\texttt{skip}]\!]\, s = s$;
- $[\![y := e]\!]\, s = s[y \leftarrow [\![e]\!]\, s]$;
- $[\![S_1\ ;\ S_2]\!]\, s = [\![S_2]\!]\, [\![S_1]\!]\, s$;
- $[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\, s = \begin{cases} [\![S_1]\!]\, s \text{ if } [\![b]\!]\, s \neq 0 \\ [\![S_2]\!]\, s \text{ if } [\![b]\!]\, s = 0 \end{cases}$.

For an expression e, we define $[\![e]\!] : \mathit{State} \to \mathit{Val}$ as follows:

- $[\![x]\!]\, s = s(x)$ if $x \in \mathsf{Dom}(s)$;
- $[\![f(e_1, \ldots, e_k)]\!]\, s = [\![f]\!]([\![e_1]\!]\, s, \ldots, [\![e_n]\!]\, s)$, where $[\![f]\!]$ is defined by the underlying SMC platform of the programming language, describing the computation of arithmetic blackbox functions.

6.3 Computational Circuits

Secure computation programs are transformed into computational circuits. These circuits are more general than the arithmetic and boolean circuits defined in Section 2.1. In particular, the gates can be arbitrary arithmetic blackbox operations. Instead of representing the local computation of parties, as it was done in Chapters 4 and 5, the circuits in this chapter represent the *functionality* that the parties mutually compute.

6.3.1 Circuit Definition

Given a set Var of program variables, we define a circuit that modifies the values of (some of) these variables. It consists of the set of gates G doing the computation, the mapping X that maps the input wires of G to the program variables Var, so that we can feed their valuations to the circuit, and the mapping Y that maps the variables of Var to the output wires of G, so that we may assign the new valuations, obtained from the circuit execution, to the program variables.

Definition 6.1. Let $Vname$ be the global set of wire names. Let Var be the set of program variables. A computational circuit is a triple $\mathsf{G} = (G, X, Y)$ where:

1. $G = \{g_1, \ldots, g_m\}$ for some $m \in \mathbb{N}$, where each $g \in G$ is of the form $g = (v, op, [v_1, \ldots, v_n])$ and the following holds:

   - $v \in Vname$ is a unique gate identifier;
   - op is the operation that the gate computes, i.e. an arithmetic blackbox function of the SMC platform;
   - $[v_1, \ldots, v_n]$ for $v_i \in Vname$ is the list of the arguments to which the operation op is applied when the gate is evaluated.

   Let $V := \{v \in Vname \mid \exists op, \vec{v} : (v, op, \vec{v}) \in G\}$.

   Let $W := \{v \in Vname \mid \exists u, op, \vec{v} : \ (u, op, \vec{v}) \in G, v \in \vec{v}\}$.

   The set of *input wires* of G is defined as $I(\mathsf{G}) := W \setminus V$. The set of all wires of G is defined as $V(\mathsf{G}) := V \cup W$.

2. $X : I(\mathsf{G}) \to Var$ assigns to a wire $v \in I(\mathsf{G})$ the variable $X(v)$.

3. Y is a mapping whose range defines the set of *output wires* $O(\mathsf{G}) \subseteq V$. $Y : Var \to O(\mathsf{G})$ assigns to a variable $y \in Var$ the wire $Y(y)$.

As a part of the definition, the directed graph induced by the input/output relations between the gates should be acyclic. The gates of G are unique, i.e. if $(u_1, op, [v_1, \ldots, v_n]) \in G$, and $(u_2, op, [v_1, \ldots, v_n]) \in G$, then $u_1 = u_2$.

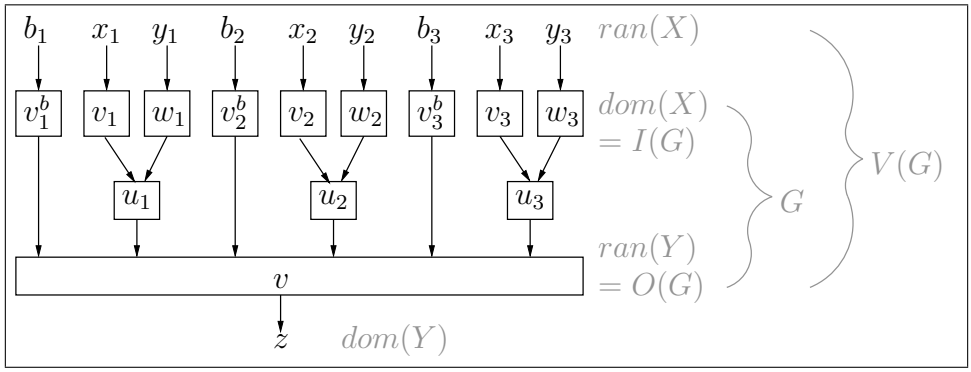


Figure 6.2: Example of a program circuit

We use $\mathcal{G}$ to denote the set of all circuits defined in this way.

In order to easily switch between the sets of G and $V(\mathsf{G})$, we define a function $\mathsf{gate}^{\mathsf{G}} : V(\mathsf{G}) \to G$ such that

$$\mathsf{gate}^{\mathsf{G}}(v) = \begin{cases} (v, op, \vec{v}) \text{ if } \exists op, \vec{v} : \ (v, op, \vec{v}) \in G \ , \\ \bot \text{ otherwise.} \end{cases}$$

Since gate names are unique, the inverse function $\mathsf{gate}^{\mathsf{G}^{-1}}$ is well-defined.

The circuits that we work on may contain gates whose operation is the *oblivious choice*. Such gates are introduced while transforming out private conditionals. An oblivious choice gate is defined as $(v, oc, [b_1, v_1, \ldots, b_n, v_n])$, and it returns the output of $\mathsf{gate}^{\mathsf{G}}(v_i)$ iff the output of $\mathsf{gate}^{\mathsf{G}}(b_i)$ is 1. If there is no such b_i, it outputs 0. It works on the assumption that at most one $\mathsf{gate}^{\mathsf{G}}(b_i)$ outputs 1. This assumption needs to be ensured by the transformation that constructs a circuit from a program.

Example 6.1. Let $z, (x_i, y_i, b_i)_{i\in[3]} \in Var$. Let the value of z be chosen obliviously from $x_1 * y_1$, $x_2 * y_2$, $x_3 * y_3$ according to the choice bits b_1, b_2, b_3. The circuit corresponding to this program, depicted in Figure 6.2, would be defined as

- $G = \{(u_1, *, [v_1, w_1]), (u_2, *, [v_2, w_2]), (u_3, *, [v_3, w_3]),$
  $(v, oc, [v_1^b, u_1, v_2^b, u_2, v_3^b, u_3])\}$;
- $X = \{v_1 \leftarrow x_1, v_2 \leftarrow x_2, v_3 \leftarrow x_3,$
  $w_1 \leftarrow y_1, w_2 \leftarrow y_2, w_3 \leftarrow y_3,$
  $v_1^b \leftarrow b_1, v_2^b \leftarrow b_2, v_3^b \leftarrow b_3\}$;
- $Y = \{z \leftarrow v\}$.

6.3.2 Circuit Evaluation

First of all, we define the circuit evaluation on its inputs, without treating it as a part of a program.

Definition 6.2. Let $W : I(\mathsf{G}) \to \mathit{Val}$ be an arbitrary valuation of the input wires of G. Let $u \in V(\mathsf{G})$, and let $[\![G]\!] : (I(\mathsf{G}) \to \mathit{Val}) \to V(\mathsf{G}) \to \mathit{Val}$ evaluate gates w.r.t. certain input valuation. We define $[\![G]\!]\, W$ inductively on $|G|$.

- $[\![\emptyset]\!]\, W\, u = W(u)$, which is correct since $u \in V(\mathsf{G}) = I(\mathsf{G}) \cup \emptyset = I(\mathsf{G})$.
- $[\![G \cup (v, f, [v_1, \ldots, v_n])]\!]\, W\, u$
  $= (u \neq v)\ ?\ [\![G]\!]\, W\, u\ :\ [\![f]\!]([\![G]\!]\, W\, v_1, \ldots, [\![G]\!]\, W\, v_n)$.

In the beginning of Section 6.3, we have defined a special *oblivious choice* gate. We now define its evaluation.

Definition 6.3. Let $b_1, \ldots, b_n \in V(\mathsf{G})$ be such that, for any input wire valuation $W : I(\mathsf{G}) \to \mathit{Val}$, $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i \in \{0, 1\}$, and $\forall i :\ [\![G]\!]\, W\, b_i \in \{0, 1\}$. The output of an *oblivious choice gate* $(v, oc, [b_1, v_1, \ldots, b_n, v_n])$ is defined as

$$[\![G]\!]\, W\, v = \sum_{i=1}^{n} ([\![G]\!]\, W\, b_i) \cdot ([\![G]\!]\, W\, v_i)\ .$$

In this definition, the sum $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i$ should belong to the set $\{0, 1\}$. Alternatively, we could more strictly define $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i = 1$. This would allow us to treat one of the choices as the *default* choice that is the negation of all the other choices. The reason why we allow $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i = 0$ is that, since we use the weakest precondition of gates for making the choice (we will define it formally in Section 6.4.1), it may happen that all the preconditions of the fused gates are false. In this case, the output of the oblivious choice gate does not matter anymore, because later there will be some other oblivious choice gate that drops it. Hence it does not matter whether it outputs 0 or some particular v_i. Therefore, one of the choices is allowed to be set to a default choice anyway, and there is no difference whether we allow $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i \in \{0, 1\}$ or just $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i = 1$. The first option just makes the presentation simpler.

Since we use the circuit evaluation as a part of the program execution, we must translate it to a program statement. Let v be a name of a circuit wire. We extend the syntax of the program with new types of statements.

$$\begin{array}{rcl} \mathit{exp} & ::= & \texttt{eval (}\ \mathit{gate}^*\ \texttt{, (}x\texttt{,}\ v\texttt{)}^*\ \texttt{, (}\ x\texttt{,}\ v\ \texttt{)}^*\ \texttt{)} \\ \mathit{gate} & ::= & \texttt{(}\ v\ \texttt{,}\ f\ \texttt{, [}\ v^*\ \texttt{])} \end{array}$$

The statement $\texttt{eval}(G, X, Y)$ evaluates the gates G, where X assigns the input values to the input wires $I(\mathsf{G})$ of G, and Y defines the set of output wires $O(\mathsf{G})$ from which the values have to be eventually taken. The gates of G are evaluated according to the definition of $[\![G]\!]$.

The semantics of the evaluation statement is defined as

$$[\![\texttt{eval}(G, X, Y)]\!]\, s = \texttt{upd}(Y \circ [\![G]\!](s \circ X), s) \ ,$$

where

$$\texttt{upd}(s', s) = x \in \mathsf{Dom}(s') \ ?\ s'(x) \ :\ s(x)$$

is the result of updating the state s with the variable valuations of some other state s'. In this way, $[\![\texttt{eval}(G, X, Y)]\!]$ modifies the state s in such a way that each variable $y \in \mathsf{Dom}(Y)$ is evaluated with the output of the gate $Y(y)$, where the valuations of the input gates are taken from s. As a shorthand notation, we write $\texttt{eval}(\mathsf{G}) = \texttt{eval}(G, X, Y)$ for $\mathsf{G} = (G, X, Y)$.

The circuits can be composed by attaching the output wires of one circuit to the input wires of another circuit. The composition of circuits as syntactic objects and the semantics preservation proof of this operation are given in Section 6.5.1.

6.3.3 Transforming a Program to a Circuit

We need to transform the private conditional statements of the initial imperative language to a circuit. Intuitively, each assignment $y := f(x_1, \ldots, x_n)$ of the initial program can be viewed as a single circuit computing a set of gates G defined by the description of f on inputs $x_1, \ldots, x_n$, where X maps the input wires of the circuit to the variables $x_1, \ldots, x_n$, and Y maps y to the output wire of the circuit. A sequence of assignments is put together into a single circuit using circuit composition.

If the program statement is not an assignment but a private conditional statement, then all its branches are first transformed to independent circuits (G_i, X_i, Y_i). The value of each variable y is then selected obliviously among $Y_i(y)$ as $y := \sum_i b_i Y_i(y)$, where b_i is the condition of executing the i-th branch. So far, the transformation is similar to the related work [106, 88], and the only difference is that we construct a computational circuit at this point.

Formally, we need to define a transformation $T_P : \ prog \ \rightarrow \ prog$ that substitutes all private conditionals of the initial program with circuit evaluations. The transformation is *correct* if for any program P, $s \in State$, it holds that $[\![P]\!]\, s = [\![T_P(P)]\!]\, s$. The formal definition of T_P and the proof of its correctness are given in Section 6.5.2.

Example 6.2. An example of transforming a conditional statement P to the circuit $T_P(P)$ is given in Figure 6.3. This circuit is defined as

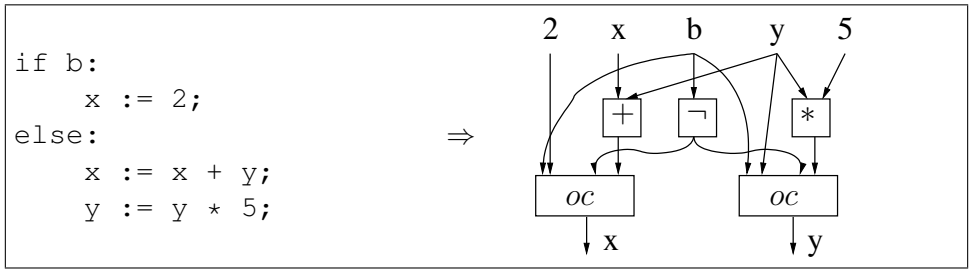


Figure 6.3: Example of program transformation

- $G = \{(u_{add}, +, [v_x, v_y]), (\bar{v}_b, \neg, [v_b]),$
  $(u_{mul}, *, [v_y, v_{const5}]), (w_x, oc, [v_b, v_{const2}, \bar{v}_b, u_{add}]),$
  $(w_y, oc, [v_b, v_y, \bar{v}_b, u_{mul}])\}$;
- $X = \{v_x \leftarrow x, v_y \leftarrow y, v_b \leftarrow b, v_{const5} \leftarrow 5, v_{const2} \leftarrow 2\}$;
- $Y = \{x \leftarrow w_x, y \leftarrow w_y\}$.

6.4 Optimization of the Circuit

The circuit G obtained from the transformation described in Section 6.3.3 may be non-optimal. Namely, it contains executions of all the branches of private conditional statements, although only one of the branches will be eventually needed. In this section, we present an optimization that eliminates excessive computations caused by the unused branches.

6.4.1 The Weakest Precondition of a Gate

Let $\mathsf{G} = (G, X, Y)$ be a computational circuit. The *weakest precondition* ϕ_v^{G} of evaluation of a gate $g = \mathsf{gate}^{\mathsf{G}}(v) \in G$ is a boolean expression over the conditional variables, such that $\phi_v^{\mathsf{G}} = 1$ iff the result of evaluating gate g is needed for the given valuations of conditional variables.

Definition 6.4 (used gate)**.** Let $\mathsf{G} = (G, X, Y)$ be a circuit. Let $s \in State$. For a wire $v \in V(\mathsf{G})$, we define a predicate $\mathsf{used}(v, s)$ as follows.

1. $\mathsf{used}(v, s) = 1$ for $v \in O(G)$ for any $s \in State$, i.e. all outputs are used.
2. If $\mathsf{used}(v, s) = 1$, $(v, op, \vec{a}) \in G$ for $op \neq oc$, then $\mathsf{used}(w, s) = 1$ for all $w \in \vec{a}$, i.e. all inputs of a used non-oc gate are also used.
3. If $\mathsf{used}(v, s) = 1$, and $(v, oc, [b_1, v_1, \ldots, b_n, v_n]) \in G$, then $\mathsf{used}(b_i, s) = 1$ for all $i \in \{1, \ldots, n\}$ (the choice conditions b_i), and $\mathsf{used}(v_j, s) = 1$ for $[\![G]\!]\,(s \circ X)\,b_j = 1$ (the choice that the oc gate makes).

Definition 6.5 (the weakest precondition (WP))**.** In the circuit $\mathsf{G} = (G, X, Y)$, the weakest precondition ϕ_v^{G} of the wire $v \in V(\mathsf{G})$ is a boolean expression over $V(\mathsf{G})$ such that $[\![\phi_v^{\mathsf{G}}]\!]\, s = 1$ iff $\mathsf{used}(v, s) = 1$, where the semantics of a boolean expression is defined as $[\![\phi_v^{\mathsf{G}}]\!]\, s := [\![u := \phi_v^{\mathsf{G}}]\!]\, s\, u$.

We propose an algorithm for computing the weakest preconditions of all the gates of a circuit. Let $BF(V)$ denote the set of all boolean formulae over a set of variables V. The algorithm constructs a mapping $\phi : V(\mathsf{G}) \to BF(V(\mathsf{G}))$, such that $\phi(v) = \phi_v^{\mathsf{G}}$. The construction of ϕ is given in Algorithm 1.

The function $\mathsf{process}(v, \phi_{in})$ takes a wire $v \in V(\mathsf{G})$ and some initially known overestimation of $\phi_{in} \in BF(V(\mathsf{G}))$ of v, which is in general the weakest precondition of some of the v's successors. This function returns a boolean formula that a wire $v \in V$ outputs, decomposed to boolean operations as far as possible. If the decomposition is impossible (for example, the gate operation is not a boolean operation, or it is an input wire), it just returns v. As a side effect, it updates the definition of function ϕ, and also the auxiliary function ψ that is used to remember the outcome of $\mathsf{process}(v, \phi_{in})$ in order to avoid processing the same wire multiple times.

We start from running process on each final output gate of G (lines 2-3). The first precondition that we propagate is 1. This means that there are no conditional constraints on v yet. Since the graph G is actually a statement of a larger program, it may happen that, instead of 1, there is a more precise condition that is coming from some public variable.

If v has already been visited ($\phi(v) \neq \bot$), it means that we have found another computational path that uses v. The condition of executing that path is ϕ_{in}, and we have already found another path with condition $\phi_v = \phi(v)$ before. Since both conditions are sufficient for forcing the computation of v, we update $\phi(v) \leftarrow \phi_v \vee \phi_{in}$, and return $\psi(v)$ that we have already computed before (lines 7-8).

If v has not been visited yet ($\phi(v) = \bot$), then, since ϕ_{in} is the weakest precondition of one of the computational paths that use v, we initialize $\phi(v) \leftarrow \phi_{in}$ on line 10. If $\mathsf{op}^{\mathsf{G}}(v) = \wedge$, we compute the outputs ψ_{out}^1 and ψ_{out}^2 of its arguments, and return $\psi_{out}^1 \wedge \psi_{out}^2$ (lines 13-16). We do it analogously for $\vee$.

The precondition ϕ_{in} will be propagated to both arguments as $\phi(v)$. It is important that here $\phi(v)$ is passed not by value, but by reference, and in the case $\phi(v)$ gets updated after v will be reached via some other branch on further steps, this update will be propagated to all its predecessors.

In the case of *oc*, we process the arguments in pairs (b, a), where b is the condition and a is the choice. The conditions are just processed recursively as $b' \leftarrow \mathsf{process}(b, \phi(v))$. However for choices we have to extend $\phi(v)$ with the output of the corresponding condition as $a' \leftarrow \mathsf{process}(a, \phi(v) \wedge b')$, since b' adds an additional restriction on the precondition of a (lines 19-23).

Algorithm 1: WP finds the weakest preconditions of all wires of G

Data: A circuit $\mathsf{G} = (G, X, Y)$
Result: A mapping $\phi : V(\mathsf{G}) \to BF(V(\mathsf{G}))$ that maps a wire to its weakest precondition

begin $\mathsf{WP}(G, X, Y)$
1 $\quad \phi \leftarrow \{\}; \psi \leftarrow \{\};$
2 $\quad$ **foreach** $v \in O(\mathsf{G})$ **do**
3 $\quad\quad \mathsf{process}(v, 1)$
4 $\quad$ **return** ϕ

begin $\mathsf{process}(v, \phi_{in})$
5 $\quad \phi_v \leftarrow \phi(v);$
6 $\quad$ **if** $\phi_v \neq \bot$ **then**
7 $\quad\quad \phi(v) \leftarrow (\phi_{in} \vee \phi_v)$
8 $\quad\quad$ **return** ψ_v
9 $\quad$ **else**
10 $\quad\quad \phi(v) \leftarrow \phi_{in}$
11 $\quad\quad$ **switch** $\mathsf{op}^{\mathsf{G}}(v)$ **do**
12 $\quad\quad\quad$ **case** *bop* **do** // $bop \in \{\wedge, \vee\}$
13 $\quad\quad\quad\quad [a_1, a_2] \leftarrow \mathsf{args}^{\mathsf{G}}(v)$
14 $\quad\quad\quad\quad \psi^1_{out} \leftarrow \mathsf{process}(a_1, \phi(v))$
15 $\quad\quad\quad\quad \psi^2_{out} \leftarrow \mathsf{process}(a_2, \phi(v))$
16 $\quad\quad\quad\quad$ **return** ψ^1_{out} *bop* ψ^2_{out}
17 $\quad\quad\quad$ **case** *oc* **do**
18 $\quad\quad\quad\quad \vec{a} \leftarrow []; \vec{b} \leftarrow [];$
19 $\quad\quad\quad\quad$ **foreach** $(b, a) \in \mathsf{args}^{\mathsf{G}}(v)$ **do**
20 $\quad\quad\quad\quad\quad b' \leftarrow \mathsf{process}(b, \phi(v))$
21 $\quad\quad\quad\quad\quad a' \leftarrow \mathsf{process}(a, b' \wedge \phi(v))$
22 $\quad\quad\quad\quad\quad \vec{a} \leftarrow \vec{a} \| (a')$
23 $\quad\quad\quad\quad\quad \vec{b} \leftarrow \vec{b} \| (b')$
24 $\quad\quad\quad\quad$ **return** $\sum_{i=1}^{|\vec{a}|=|\vec{b}|} b_i \cdot a_i$
25 $\quad\quad\quad$ **otherwise do**
26 $\quad\quad\quad\quad$ **foreach** $a \in \mathsf{args}^{\mathsf{G}}(v)$ **do**
27 $\quad\quad\quad\quad\quad \mathsf{process}(v, \phi(v))$
28 $\quad\quad\quad\quad$ **return** v

The output of the oc gate is defined by the line 24. The output condition of an oc gate is $\sum_{i=1}^{|\vec{a}|=|\vec{b}|} b_i \cdot a_i$, where b_i is a condition that has to be satisfied for the choice a_i to be output. For any other gate, we just process the arguments recursively (line 27), and return v on line 28.

As the result, each wire $v \in V(\mathsf{G})$ will be assigned a boolean formula $\phi_v^{\mathsf{G}} = \phi(v)$ over $V(\mathsf{G})$. For a wire that should be computed in any case, we have $\phi^{\mathsf{G}}(v) = 1$. We will never fuse gates having such output wires.

The *cost* of the weakest precondition (denoted $cost(\phi_i^{\mathsf{G}})$) is just the total cost of all the $\vee$ and $\wedge$ operations used in it, without taking into account the complexity of computing its variables (their cost is estimated separately). To improve the optimization, we could try to rearrange $\vee$ and $\wedge$ operations in each ϕ_i^{G} to make the cost optimal. This is not in scope of this thesis.

The correctness of Algorithm 1 is proven in Section 6.5.3.

6.4.2 Informal Description of the Optimization

Let $g_1 = (v_1, op, [x_1^1, \ldots, x_n^1]), \ldots, g_k = (v_k, op, [x_1^k, \ldots, x_n^k]) \in G$ be some gates. Let $\phi_{v_1}^{\mathsf{G}}, \ldots, \phi_{v_k}^{\mathsf{G}}$ be mutually exclusive. This happens for example if each g_i belongs to a distinct branch of a set of nested conditional statements. In this case, we can *fuse* the gates $g_1, \ldots, g_k$ into a single gate g that computes the same operation op, choosing each of its inputs x_j obliviously among $x_j^1, \ldots, x_j^k$. This introduces n new oblivious choice gates, but leaves just one gate g computing op.

Example 6.3. Let a comparison operation $(==)$ be located in both the `if`-branch, and the corresponding `else`-branch. Let the gates be $(z_1, ==, [x_1, y_1])$ and $(z_2, ==, [x_2, y_2])$. Let $b \in V(\mathsf{G})$ be the wire whose value is the condition of the `if`-branch, and let $\bar{b} \in V(\mathsf{G})$ be the wire whose value is b's negation (which is the condition of the `else`-branch). Since the branches can never be executed simultaneously, we may replace these gates with $(x, oc, [b, x_1, \bar{b}, x_2])$, $(y, oc, [b, y_1, \bar{b}, y_2])$, and $(z, ==, [x, y])$. All the references to z_1 and z_2 in the rest of the circuit are now substituted with the reference to z. The transformation is depicted in Figure 6.4.

We now describe different steps of this optimization. Let $n = |V(\mathsf{G})|$, $m = |G|$.

Preprocessing First, we find the set U of all pairs of gates that can never be evaluated simultaneously. For each gate g_i, find the weakest precondition ϕ_i^{G} that be must true for g_i to be evaluated. Define

$$U = \{(g_i, g_j) \mid g_i, g_j \in G, \phi_i^{\mathsf{G}} \wedge \phi_j^{\mathsf{G}} \text{ is unsatisfiable}\} \ .$$

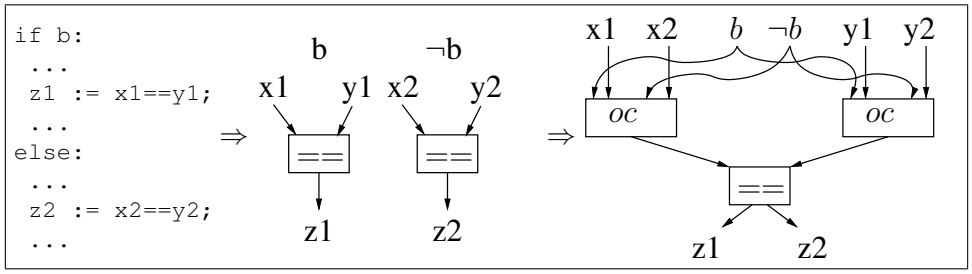


Figure 6.4: An example of gate fusing

Although there is no efficient algorithm for solving the unsatisfiability problem, in practice, it suffices to find only a subset $U' \subseteq U$. It makes the optimization less efficient (we do not fuse as many gates as we could), but nevertheless correct.

For each formula ϕ_i^{G}, we need to construct the circuit that computes the value of ϕ_i^{G}. Depending on how exactly ϕ_i^{G} is computed, evaluating this circuit may in turn have some cost. Each ϕ_i^{G} is represented by a boolean formula over the conditions of the `if`-statements of the initial program, which can be read out from G by observing its oblivious choice gates. The additional $\vee$ and $\wedge$ gates are needed in the cases where a gate is located inside several nested `if`-statements (needs $\wedge$), or it is used in several different branches (needs $\vee$).

Plan We partition the gates into sets C_k, planning to leave only one gate from C_k after the optimization. The following conditions should hold:

- $\forall g_i, g_j \in C_k : \; g_i \neq g_j \implies (g_i, g_j) \in U$: we put together only mutually exclusive gates, so that indeed at most one gate of C_k will actually be executed.
- $\forall g_i, g_j \in C_k : \; op_i = op_j$: only the gates that compute the same operation are put together.
- Let $E := \{(i,j) \mid \exists k, \ell : g_k \in C_i, g_\ell \in C_j$, and g_k is an immediate predecessor of g_ℓ in $G\}$. In this way, if $(i,j) \in E$, then C_i should be evaluated strictly before C_j. We require that the graph $(\{k\}_{k\in[m]}, E)$ is acyclic. Otherwise, we might get the situation where some gates of C_j have to be computed necessarily before C_i, and at the same time some gates of C_i should be computed necessarily before C_j, so evaluating all the gates of C_i at once would be impossible.

If we consider U as edges, we get that C_k form a set of disjoint cliques in the graph. A possible fusing of gates into a clique is shown in Figure 6.5, where the gray lines connect the pairs $(g_i, g_j) \in U$, and the dark gates are treated as a single clique.

Transformation The plan gives us a collection of sets of gates C_j, each having gates of certain operation op_j. Consider any set $C_j = \{g_1, \ldots, g_{m_j}\}$. Let the inputs of the gate g_i be $x_1^i, \ldots, x_n^i$. Let b_i be the wire that outputs the value of ϕ_i^{G}. For all $\ell \in [n]$, introduce n new oblivious choice gates $(v_\ell, oc, [b_1, x_\ell^1, \ldots, b_{m_j}, x_\ell^{m_j}])$. Add the new gate $(g, op_j, [v_1, \ldots, v_n])$. Discard all the gates g_i. If any gate in the rest of the circuit has used any g_i as an input, substitute it with g instead. We may additionally omit any oblivious choice in the graph if there is just one option to select from.

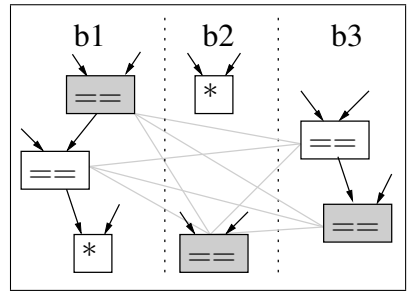


Figure 6.5: Fusing gates into cliques

For each oc gate that has been already present in the circuit, check how many distinct inputs it has. It is possible that some inputs have been fused into one due to belonging to the same clique. In this case, it may happen that the oc gate is left with a single choice, and since $(v_\ell, oc, [b_i, x_\ell^i])$ just returns x_ℓ^i, its cost is 0.

The Cost Our goal is to partition the cliques in such a way that the *cost* of the resulting circuit is minimal. Each gate operation corresponds to some SMC protocol that requires some amount of bits to be communicated between the parties. We choose the total number of communicated bits as the cost. Since this metric is additive, we may easily estimate the total cost of the circuit by summing up the communicated bits of the gates. The particular costs of the gates depend on the chosen SMC platform. We note that introducing intermediate oblivious choices may increase the number of rounds. We need to be careful, since increasing the number of rounds may make executing the circuit with a lower communication cost actually take more time.

6.4.3 Notation

Shorthand Notation Let $\mathsf{G} = (G, X, Y)$, $(g, op, [v_1, \ldots, v_n]) \in G$. We introduce the following shorthand notation:

- $\mathsf{op}^{\mathsf{G}}(g) = op$;
- $\mathsf{args}^{\mathsf{G}}(g) = [v_1, \ldots, v_n]$;
- $\mathsf{arity}^{\mathsf{G}}(g) = n$;

For shortness of notation, we write $g \in G$ in place of $\mathsf{gate}^{\mathsf{G}}(g) \in G$.

Gate and Clique Enumeration Without loss of generality, $V(\mathsf{G}) = \{1, \ldots, n\}$. This ordering allows to easily define the constraints for linear program variables.

The number of cliques varies between 1 (if all the gates are fused together into one) and $|G|$ (if each gate is a singleton clique). For simplicity, we assume that we always have exactly $|G|$ cliques, and some of them may just be left empty. We denote the clique $\{i_1, \ldots, i_k\}$ by C_j, where $j = \min(i_1, \ldots, i_k)$ is the *representative* of the clique C_j. Without loss of generality, let the representative be the only gate that is left of C_j after the fusing.

Direct and Conditional Predecessors In order to ensure that the optimized circuit contains no cycles, we need to remember which gates have been predecessors of each other. We define the following auxiliary predicates that can be easily computed from the initial circuit.

- $\mathsf{pred}^{\mathsf{G}}(i, k) = 1$ iff $k \in \mathsf{args}^{\mathsf{G}}(i)$;
- $\mathsf{cpred}^{\mathsf{G}}(i, k) = 1$ iff $k \in \phi^{\mathsf{G}}(i)$.

The predicate $\mathsf{pred}^{\mathsf{G}}(i, k)$ is true if k is an immediate predecessor of i in G. The predicate $\mathsf{cpred}^{\mathsf{G}}(i, k)$ is true if k is used to compute the weakest precondition of i. This means that k does not have to be computed strictly before i in general. However, if i is fused with some other gate, we will need the value of k for computing the oblivious choice of the arguments of i, and in this case k has to be computed strictly before i. In this way, k is a predecessor of i on the condition that i is fused with at least one other gate.

Gates that can be fused We define an auxiliary predicate that denotes which gates are allowed to be fused:

$$\mathsf{fusable}^{\mathsf{G}}(i, j) = 1 \text{ iff } (i = j) \vee (\phi^{\mathsf{G}}(i) \wedge \phi^{\mathsf{G}}(j) \text{ is unsatisfiable}) \ .$$

Although there exists no efficient algorithm for computing unsatisfiability in general, we may allow some algorithm that provides false negatives. This results in having $\mathsf{fusable}^{\mathsf{G}}(i, j) = 0$ for gates that could have actually been fused, and hence the final solution may be non-optimal, but nevertheless correct.

Since fusing forces all the gate arguments to become chosen obliviously, all the inputs of a fused gate in general become private (unless there was just one choice for some public input). Depending on the SMC platform and the particular operation, this may formally change the gate operation. Some operations still retain the same cost, while some gates may increase their cost significantly if some of their public inputs become private. Moreover, it may happen that the new operation is not supported by the SMC platform at all. We define $\mathsf{fusable}^{\mathsf{G}}(i, j) = 0$ for the gates that have any public inputs, and whose cost depends on their privacy.

6.4.4 Subcircuits as Gates

Let $\mathsf{G} = (G, X, Y) \in \mathcal{G}$. In some cases, there are obvious repeating patterns of gates in G which could be treated as a single gate. Uniting them into one gate would reduce the total number of gates involved in the optimization, increasing its efficiency. Unlike the process of *fusing* gates, where a set of gates is replaced with a single gate, the process of *uniting* keeps all the gates and just treats their set as a single entity.

We propose a particular algorithm for partitioning G into a set of disjoint subcircuits. The algorithm constructs the set of subcircuits iteratively, starting from the initial set of gates A_0, and on each iteration constructing A_{n+1} by putting together certain subcircuits of A_n. Let $S \in A_n$ denote a subcircuit of A_n, and let $S' \sqsubseteq S$ denote that $S' \in A_n$ has become a part of $S \in A_{n+1}$. Treating S as a set of gates, we write $g \in S$ for a gate $g \in G$ that has been united into S. We may extend the notion of an *argument* from single gates to subcircuits as $S' \in \mathsf{args}^{A_n}(S) \iff \exists g \in S, g' \in S' : \; g' \in \mathsf{args}^{\mathsf{G}}(g)$.

Let the subcircuits S and S' be called *isomorphic* if there exists a bijection between $V(S)$ and $V(S')$ preserving the circuit structure and the gate operations. Such isomorphisms are simple to find due to the inputs of all gates being ordered. Let $\mathsf{count}(S, A_n)$ be the number of elements in A_n that are isomorphic to S. The sets A_n are defined inductively as follows.

$$
\begin{aligned}
A_0 &:= \{\{g\} \mid g \in G\} \;; \\
A'_{n+1} &:= \{S \cup \bigcup \{S_i \in \mathsf{args}^{A_n}(S) \mid \neg\exists T \in A'_{n+1} : S_i \subseteq T, \\
&\qquad\qquad \neg\exists S' \neq S \in A_n : \; S_i \in \mathsf{args}^{A_n}(S')\} \\
&\quad \mid S \in A_n\} \;; \\
A_{n+1} &= \{S \mid \mathsf{count}(S, A'_{n+1}) \geq 2\} \\
&\quad \cup \{S' \mid S \in A'_{n+1}, \mathsf{count}(S, A'_{n+1}) = 1, S' \sqsubseteq S\} \;.
\end{aligned}
$$

Let us explain the intuition behind this definition. We start from the initial set of gates G, treating each gate as a singleton subcircuit. On each iteration, we extend each subcircuit with its argument subcircuits that are not used as arguments by any other subcircuits; this is the interpretation of $\neg\exists S' \neq S \in A_n : \; S_i \in \mathsf{args}^{A_n}(S')$. We want the subcircuits to be disjoint, and hence if some $S_i \in A_n$ has already been extended on this iteration, then we are not trying to use S_i to extend some other $S \in A_n$; that is how the condition $\neg\exists T \in A'_{n+1} : S_i \subseteq T$ should be interpreted.

If any subcircuit S occurs only once in A'_{n+1}, then S is no longer interesting for our optimization, since it cannot be fused with any other gate. Therefore, after each iteration we may leave only those subcircuits of A'_{n+1} that occur at least twice. Each $S \in A'_{n+1}$ that occurs only once is decomposed back to its subcircuits $S' \sqsubseteq S$, $S' \in A_n$. After doing all these decompositions, we get A_{n+1}.

Such definition of A_n allows us to define args^{A_n} inductively as follows.

$$\begin{aligned}\mathsf{args}^{A_0}(\{g\}) &:= \{\{a\} \mid a \in \mathsf{args}^{\mathsf{G}}(g)\} \ ; \\ \mathsf{args}^{A_{n+1}}(S) &:= \{S_i \mid S' \sqsubseteq S, S_i \in \mathsf{args}^{A_n}(S'), S_i \not\sqsubseteq S\} \ .\end{aligned}$$

In other words, the arguments of a subcircuit S is any subcircuit S_i that has been an argument of some $S' \sqsubseteq S$ on the previous iteration, and that has not become the part of S on the current iteration.

Let $\mathsf{Subcircuit}(G, n)$ be a function computing the set of subcircuits A_n for the given G and n. After composing the subcircuits in this way, we are only allowed to use the final outputs of the subcircuits. The outputs of the gates that are swallowed by a subcircuit can only be used inside that subcircuit. Hence we need to prove that the set of new gates A_n still does exactly the same computation as G. This is formally stated and proven in Section 6.5.4. In this way, the optimization proposed in further subsections can be applied to single gates as well as to the partitions formed by the function Subcircuit.

6.4.5 Simple Greedy Heuristics

We investigate some simple greedy heuristics for our task. Let $\overline{\mathsf{pred}}^{\mathsf{G}}$ and $\overline{\mathsf{cpred}}^{\mathsf{G}}$ be the transitive closures of the predicates $\mathsf{pred}^{\mathsf{G}}$ and $\mathsf{cpred}^{\mathsf{G}}$ respectively.

- $\overline{\mathsf{pred}}^{\mathsf{G}}(i,j) = 1$ for all i, j s.t $\mathsf{pred}^{\mathsf{G}}(i,j) = 1$;
- $\overline{\mathsf{pred}}^{\mathsf{G}}(i,j) = 1$ if $\exists k : \ \mathsf{pred}^{\mathsf{G}}(i,k) \wedge \mathsf{pred}^{\mathsf{G}}(k,j)$.

The predicate $\overline{\mathsf{cpred}}^{\mathsf{G}}$ will be extended a bit more, taking into account $\mathsf{pred}^{\mathsf{G}}(k,j)$.

- $\overline{\mathsf{cpred}}^{\mathsf{G}}(i,j) = 1$ for all i, j s.t $\mathsf{cpred}^{\mathsf{G}}(i,j) = 1$
- $\overline{\mathsf{cpred}}^{\mathsf{G}}(i,j) = 1$ if $\exists k : \ \mathsf{cpred}^{\mathsf{G}}(i,k) \wedge \mathsf{cpred}^{\mathsf{G}}(k,j)$;
- $\overline{\mathsf{cpred}}^{\mathsf{G}}(i,j) = 1$ if $\exists k : \ \mathsf{cpred}^{\mathsf{G}}(i,k) \wedge \mathsf{pred}^{\mathsf{G}}(k,j)$.

We need the last item since all the predecessors j of a conditional predecessor k of i will also become predecessors of i, if i gets fused into some clique. We note that $\overline{\mathsf{cpred}}^{\mathsf{G}}$ gives an overestimation, since if i is fused, then j is not necessarily a predecessor of i if k is not fused. This reduces the number of allowed fusings in the circuit, but nevertheless does not introduce any incorrect solutions.

The greedy algorithms work according to the following outline. First of all, the gates G are grouped by their operation into subsets

$$Gs = \{G^F \mid F \text{ is a gate operation}, G^F = \{g \in G \mid \mathsf{op}^{\mathsf{G}}(g) = F\}\} \ .$$

The subsets G^F are sorted according to the cost of F, so that more expensive gates come first. The subsets are turned into cliques one by one, starting from the most expensive operation. A clique C_k is formed only if it is valid and is not in contradiction with already formed cliques, i.e:

- any two gates $g_i, g_j \in C_k$ satisfy $\mathsf{fusable}^{\mathsf{G}}(g_i, g_j) = 1$;
- no gate $g_i \in C_k$ has already been included into some other clique;
- C_k does not introduce cycles.

This process is described by the function Greed given in Algorithm 2. The set of gates G is partitioned into a collection of subsets Gs on line 1. The obtained subsets G^F are sorted according to $cost(F)$ on line 2, where the function $\mathsf{sort}(\vec{x}, i, j)$ can be any algorithm that sorts a list of tuples $\vec{x}$ according to their i-th components, leaving behind the j-th component of each tuple. After that, the algorithm starts fusing the gates into cliques, starting from the most expensive gates, adding a clique only if it is not in contradiction with already formed cliques. The function call $\mathsf{FuseX}(G, Cs)$ on line 5 returns a partitioning of gates G to cliques, choosing a particular strategy X, which can be any of the Algorithms 4, 5, 6.

The function $\mathsf{goodClique}(C, Cs)$ (Algorithm 3) checks whether a clique C is not in contradiction with already formed cliques Cs, and it also assigns a *level* to each good clique – the round on which the gate should be evaluated. In this way, if some predecessor of a gate is not computed on a strictly earlier round than the gate itself, then we have a cycle in the circuit, so something must be wrong with the formed cliques. For this, the *maximal level of all the predecessors* of C and the *minimal level of all the successors* of C are computed (the sets L^{pred} and L^{succ} on lines 3-4). It is checked whether each gate of C can be assigned a level strictly between these two (lines 5-7). For conditional predecessors $\overline{\mathsf{cpred}}^{\mathsf{G}}$, we additionally check whether the size of the successor clique is larger than 1, i.e. whether the conditional predecessor actually becomes a predecessor after fusing. The number N on line 6 is just an upper bound on the number of levels, and it could be as well assigned to 1 since we may treat levels as rational numbers. The function level is global, and it is used to memorize the levels of the gates that have already been put into cliques. Initially, $\mathsf{level}(k) = \bot$ for all $k \in G$, and we assume that $\min(\bot, n) = \max(\bot, n) = n$ for all $n \in \mathbb{N}$.

The particular strategies of extracting a clique are given in the Algorithms 4, 5, 6. These algorithms are not optimized, and rather explain the used strategies.

Largest Cliques First. In Algorithm 4, we are trying to fuse into one clique as many gates as possible before proceeding with the other cliques. The task of finding one maximum clique is NP-hard, and so we just generate some bounded amount of maximal cliques, taking the largest of them. Extracting one clique is done by the function largestClique. Fixing some gate as a starting point, we

Algorithm 2: Greed fuses mutually exclusive gates of G into cliques

Data: A set of gates G
Result: A set of cliques Cs of gates G
1 $Gs \leftarrow \{(cost(F), G^F) \mid F \text{ is gate op}, G^F = \{g \mid g \in G, \mathsf{op}^{\mathsf{G}}(g) = F\}\}$;
2 $Gs \leftarrow \mathsf{sort}(Gs, 0, 1)$;
3 $Cs \leftarrow \emptyset$;
4 **foreach** $G^F \in Gs$ **do**
5 $\quad Cs \leftarrow Cs \cup \{\mathsf{FuseX}(G^F, Cs)\}$;
6 **return** Cs;

sequentially try to add each other gate, checking whether we still have a clique, and whether it valid w.r.t. already existing cliques (line 6). Having done it for each gate as a starting point, the largest clique is returned (line 7), where $\mathsf{max}(\vec{x}, i, j)$ can be any algorithm that returns the j-th component of the element of $\vec{x}$ whose i-th component is the largest. If such C does not exist, then we have some inconsistency in cliques, and $\perp$ is returned on line 7. After extracting a clique C on line 3 of function Fuse1, add C to the set of already formed cliques Cs' of G, and proceed with extracting largest cliques from the remaining gates $G \setminus C$.

Pairwise Merging. In Algorithm 5, we first try to fuse the gates pairwise, and only after the pairs are formed, we proceed fusing the obtained cliques in turn pairwise, until the total number of cliques cannot be decreased anymore. The function matching just takes the first valid matching that it succeeds to construct, that is not in contradiction with Cs. The choice of G_1 and G_2 on line 7 is non-deterministic, and it is only important that $G_1 \neq G_2$ will be iterated before $G_1 = G_2$ to avoid trivial solutions. If such G_1 and G_2 do not exist, then we have some inconsistency in cliques, and $\perp$ is returned on line 10.

Pairwise Merging with Maximum Matching. Algorithm 6 is very similar to Algorithm 6, and the only difference is that it finds the *maximum* matching on each step. We assume that the function $\mathsf{someMatching}(Gs)$ generates all possible matchings of Gs (in contrast, matching of Algorithm 5 takes the first valid solution it finds). For better efficiency, it is sufficient to generate only *maximal* matchings, but to guarantee termination, at least the partitioning to singleton gates should be included. On line 7, the largest valid matching is taken. If there is no valid matching, then we have some inconsistency in cliques, and $\perp$ is returned on line 7.

It is easy to see that, unless $\perp$ is returned, all the formed cliques have passed goodClique check w.r.t. each other, and $\mathsf{level}(k)$ has been assigned to each gate $k \in G$. We prove in Section 6.5.5 that if the initial circuit is properly constructed, then $\perp$ will never be returned. In particular, after a clique has been fixed, it is always possible to assign the remaining gates to cliques without backtracking.

Algorithm 3: goodClique checks if the clique is valid

Data: A clique C, and the set of already existing cliques Cs. There is a global map level, and an upper bound N on the number of levels.
Result: A bit denoting whether C is valid w.r.t. Cs
begin goodClique(C, Cs)

foreach $i, j \in C$ **do**
  if not $\mathsf{fusable}^{\mathsf{G}}(i, j)$ **then**
    return false
$L^{pred} \leftarrow \{\mathsf{level}(k) \mid i \in C, \overline{\mathsf{pred}}^{\mathsf{G}}(i, k) \vee \overline{\mathsf{cpred}}^{\mathsf{G}}(i, k) \ \wedge |C| > 1\}$;
$L^{succ} \leftarrow \{\mathsf{level}(k) \mid i \in C, \overline{\mathsf{pred}}^{\mathsf{G}}(k, i) \vee \overline{\mathsf{cpred}}^{\mathsf{G}}(k, i) \wedge k \in C', C' \in Cs, |C'| > 1\}$;
$n_1 \leftarrow \mathsf{max}(\{0\} \cup L^{pred})$;
$n_2 \leftarrow \mathsf{min}(\{N\} \cup L^{succ})$;
if $n_2 < n_1$ **then**
  return false;
foreach $i \in C$ **do**
  $\mathsf{level}(i) \leftarrow (n_1 + n_2)/2$;
return true;

Algorithm 4: Fuse1 partitions the gates into cliques

Data: A set of gates G of the same operation type
Data: The set of already existing cliques Cs
Result: Partitioning of G to cliques
begin Fuse1(G, Cs)

$Cs' \leftarrow \emptyset$;
repeat
  $C \leftarrow \mathsf{largestClique}(G, Cs \cup Cs')$;
  $G \leftarrow G \setminus C$; $Cs' \leftarrow Cs' \cup \{C\}$;
until $|G| \leq 0$;
return Cs';

begin largestClique(G, Cs)

$Cs' \leftarrow \{(|C|, C) \mid i \in G, (C \leftarrow \{i\}) \vee (C \leftarrow \{i\} \cup \{j \mid j \in G, \forall k \in C : \mathsf{fusable}^{\mathsf{G}}(k, j)\}), \mathsf{goodClique}(C, Cs)\}$;
if $Cs' = \emptyset$ **then**
  return $\perp$;
return $\mathsf{max}(Cs', 0, 1)$;

Algorithm 5: Fuse2 partitions the gates into cliques

Data: A set of gates G of the same operation type
Data: A set of already existing cliques Cs
Result: Partitioning of G to cliques
begin Fuse2(G, Cs)
1 $\quad Cs' \leftarrow \{\{g\} \mid g \in G\}$;
2 $\quad$ **repeat**
3 $\quad\quad n \leftarrow |Cs'|$;
4 $\quad\quad Cs' \leftarrow \mathsf{matching}(Cs', \emptyset, Cs)$;
$\quad$ **until** $|Cs'| \geq n$;
5 $\quad$ **return** Cs';
begin matching(Gs, Cs', Cs)
6 $\quad$ **if** $Gs = \emptyset$ **then**
$\quad\quad$ **return** Cs'
7 $\quad$ **if** $\exists G_1, G_2 \in Gs :$ goodClique$(G_1 \cup G_2, Cs \cup Cs')$ **then**
8 $\quad\quad$ **return** matching$(Gs \setminus \{G_1, G_2\}, Cs' \cup \{G_1 \cup G_2\}, Cs)$;
9 $\quad$ **else**
10 $\quad\quad$ **return** $\perp$;

Algorithm 6: Fuse3 partitions the gates into cliques

Data: A set of gates G of the same operation type
Data: A set of already existing cliques Cs
Result: Partitioning of G to cliques
begin Fuse3(G, Cs)
1 $\quad Cs' \leftarrow \{\{g\} \mid g \in G\}$;
2 $\quad$ **repeat**
3 $\quad\quad n \leftarrow |Cs'|$;
4 $\quad\quad Cs' \leftarrow \mathsf{maxMatching}(Cs', Cs)$;
$\quad$ **until** $|Cs'| \geq n$;
5 $\quad$ **return** Cs';
begin maxMatching(Gs, Cs)
6 $\quad Css \leftarrow \{(|Cs'|, Cs') \mid Cs' \leftarrow \mathsf{someMatching}(Gs),$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \forall C \in Cs' :$ goodClique$(C, Cs \cup Cs')\}$;
7 $\quad$ **if** $Css = \emptyset$ **then**
$\quad\quad$ **return** $\perp$;
8 $\quad$ **else**
$\quad\quad$ **return** max$(Css, 0, 1)$;

6.4.6 Reduction to an Integer Linear Programming Task

As an alternative to greedy algorithms, we may reduce the gate fusing task to a mixed integer linear programming (ILP) task defined in Section 2.3.6, and solve it using an external integer linear program solver such as [44].

We consider mixed integer programs of the form (2.1). Let $\mathcal{ILP}$ be the set of all mixed integer programs defined as tuples $(A, \vec{b}, \vec{c}, \mathcal{I})$. For our particular task, we define a transformation $T^{\rightarrow}_{ILP} : \mathcal{G} \rightarrow \mathcal{ILP}$, such that $T^{\rightarrow}_{ILP}(G, X, Y) = (A, \vec{b}, \vec{c}, \mathcal{I})$. In this subsection, we describe how these quantities are constructed.

In order to make integer programming solutions better comparable to greedy algorithms, we consider two levels of optimization:

- **Basic:** try to optimize only the total cost of the gates, without taking into account the oc gates.
- **Extended:** take into account the new oc gates, the weakest preconditions, and also the number of inputs of the old oc gates.

Throughout this section, we use G to refer to the initial circuit, and G' to refer to the circuit obtained after the transformation. For a clique C_j, let us denote the set of all possible choices for the ℓ-th input of the clique C_j as $\mathsf{args}^{\mathsf{G}}(C_j)[\ell] := \{k \mid i \in C_j, k = \mathsf{args}^{\mathsf{G}}(i)[\ell]\}$.

Variables

The core of our optimization are the variables that affect the cost of the transformed circuit. All these variables describe not the initial circuit G, but the transformed circuit G', although the set of variables itself is defined by G.

- $g^j_i = \begin{cases} 1, \text{if } i \in C_j \\ 0, \text{otherwise} \end{cases}$ for $i, j \in G$.
  The gate j will be the representative of C_j. Namely, $g^j_j = 1$ iff C_j is non-empty. Fixing the representative reduces the number of symmetric solutions significantly. This also allows us to compute the cost of all the cliques.
- $sc^j_\ell = |\mathsf{args}^{G'}(j)[\ell]| - 1$ for $j \in G$, $\ell \in \mathsf{arity}^{\mathsf{G}}(j)$,
  is the number of decisions to make for choosing the ℓ-th argument of C_j.
- $uc^j = |\mathsf{args}^{G'}(j)| - 1$ for $j \in G$, $\mathsf{op}^{\mathsf{G}}(j) = oc$,
  is the number of decisions to make for the gate j whose operation is oc, after some of its choices have potentially been fused together.

- $s_\ell^j = \begin{cases} 1, \text{if the } \ell\text{-th input of } j \text{ should be a new } oc \text{ gate} \\ 0, \text{otherwise} \end{cases}$
  for $j \in G$, $\ell \in \mathsf{arity}^{\mathsf{G}}(j)$.
  The variables sc_ℓ^j and s_ℓ^j allow to count for the total cost of the new oc gates introduced by the optimization.

- $u^j = \begin{cases} 1, \text{if } |\mathsf{args}^{G'}(j)| > 1 \\ 0, \text{otherwise} \end{cases}$ for $j \in G$, $\mathsf{op}^{\mathsf{G}}(j) = oc$.
  If $u^j = 0$, then the oc gate j can be removed since there is only one choice left. The variables uc^j and u^j estimate the new cost of the old oc gates.

- $b_i = \begin{cases} 1, \text{if the weakest precondition of } i \text{ is needed} \\ 0, \text{otherwise} \end{cases}$ for $i \in G$.
  Fusing the gates requires their inputs to be chosen obliviously. For that, we may need to compute the weakest preconditions of the participating gates.

We also need some variables that help to avoid cycles after fusing the gates. Similarly to greedy algorithms of Section 6.4.5, we assign to each gate the round on which it has to be evaluated.

- $\ell_j \in \mathbb{R}$ for $j \in G$ is the circuit topological level on which the j-th gate is evaluated, where all the gates with the same level are evaluated simultaneously. Each gate must have a strictly larger level than all its predecessors.

- $c_j = \begin{cases} 1, \text{if the gate } g_j \text{ is fused with some other gate} \\ 0, \text{otherwise} \end{cases}$ for $j \in G$.
  Each gate should have a strictly larger level than all its *conditional* predecessors iff it participates in a clique of size at least 2. After the gates are fused into a clique, their inputs are going to be chosen obliviously, and hence the condition will have to be known strictly *before* the fused gates are evaluated.

There will actually be some more auxiliary variables that help to establish relations between the main variables, but do not have special meaning otherwise. We will see these variables when we define constraints.

Cost Function

The cost of the resulting circuit depends on the following quantities.

- $\mathsf{C}_g = \sum_{j=1,\mathsf{op}^{\mathsf{G}}(j)\neq oc}^{|G|} cost(\mathsf{op}^{\mathsf{G}}(j)) \cdot g_j^j$ is the total cost of the cliques after fusing (except the oc gates).

- $\mathsf{C}_{oc1} = \sum_{j=1,\mathsf{op}^{\mathsf{G}}(j)=oc}^{|G|} cost(oc_{base}) \cdot u^j + cost(oc_{step}) \cdot uc^j$ is the total cost of all the old oc gates, where $cost(oc_{base})$ is the base cost of using an oc gate, and $cost(oc_{step})$ is the cost of a single choice of the oc gate.

- $\mathsf{C}_{oc2} = \sum_{j,\ell=1,1}^{|G|,\mathsf{arity}^{\mathsf{G}}(j)} cost(oc_{base}) \cdot s_\ell^j + cost(oc_{step}) \cdot sc_\ell^j$ is the total cost of all the new oc gates.

- $\mathsf{C}_b = \sum_{j=1}^{|G|} cost(\phi^{\mathsf{G}}(j)) \cdot b_j$ is the cost of all the boolean conditions needed for the new oc gates.

We may now take one of the following quantities as the cost:

- **Basic cost:** C_g, which is just the total cost of the obtained cliques (in this case, the costs of the old oc gates are included into C_g).

- **Extended cost:** $\mathsf{C}_g + \mathsf{C}_{oc1} + \mathsf{C}_{oc2} + \mathsf{C}_b$, which takes into account also the cost of the new oc gates.

This describes the full cost of the gates involved in the sum, since the bit communication metric of the gates is additive. This sum would not work if we had chosen the number of rounds as the cost.

Inequality Constraints

The constraints $A\vec{x} \leq \vec{b}$ state the relations between the variables defined in Section 6.4.6. Since $A\vec{x} \geq \vec{b}$ is equivalent to $-A\vec{x} \leq -\vec{b}$, we may as well use $\leq$, $\geq$, and $=$ relations in the constraints.

Building blocks for constraints. There are some logical statements that are used several times in the constraints. We will now describe how such statements are encoded as sets of constraints (possibly with some auxiliary variables). We also define special notations for these sets of constraints.

- **Multiplication by a bit:** $z = x \cdot y$, where $x \in \{0, 1\}$, $y, z \in \mathbb{R}$, and $C \in \mathbb{R}$ is a known upper bound on y. This can be expressed by the following set of constraints:

  - $C \cdot x + y - z \leq C$;
  - $C \cdot x - y + z \leq C$;
  - $C \cdot x - z \geq 0$.

  We denote this set of constraints by $\mathcal{P}(C, x, y, z)$.

- **Threshold:**
  $y = \begin{cases} 1 \text{ if } \sum_{x \in \mathcal{X}} x \geq A \\ 0 \text{ otherwise} \end{cases}$,
  where $\forall x \in \mathcal{X} : x \in \{0, 1\}$, $y \in \mathbb{R}$, and $A \in \mathbb{R}$ is some constant. Note that, while there are no constraints on $y \in \mathbb{R}$, the property $y \in \{0, 1\}$ should be ensured by the threshold itself. This can be expressed by the following set of constraints:

  - $\mathcal{P}(1, y, x, z_x)$ for all $x \in \mathcal{X}$, where z_x are fresh variable names;
  - $A \cdot y - \sum_{x \in \mathcal{X}} z_x \leq 0$;
  - $\sum_{x \in \mathcal{X}} x - \sum_{x \in \mathcal{X}} z_x + (A - 1)y \leq (A - 1)$.

  We denote this set of constraints by $\mathcal{F}(A, \mathcal{X}, y)$.

- **Implying inequality:** $(z = 1) \implies (x - y \geq A)$, where $z \in \{0, 1\}$, $x, y \in \mathbb{R}$, $A \in \mathbb{R}$ is some constant, and $C \in \mathbb{R}$ is a known upper bound on x, y. This can be expressed by the following constraint:

  - $(C + A) \cdot z + y - x \leq C$.

  We denote this constraint by $\mathcal{L}(C, A, x, y, z)$.

The correctness of these sets of constraints is proven in Section 6.5.6.

Basic constraints. The particular constraints defining the integer linear programming task are the following.

1. $g_i^j + g_k^j \leq 1$ for $i, k \in G$, $\neg\mathsf{fusable}^{\mathsf{G}}(i, k)$.
   If the gates are not mutually exclusive, then they cannot belong to the same clique.

2. $\sum_{j=1}^{|G|} g_i^j = 1$ for all $i \in G$.
   Each gate belongs to exactly one clique.

3. $g_i^j = 0$ if $\mathsf{op}^{\mathsf{G}}(i) \neq \mathsf{op}^{\mathsf{G}}(j)$.
   The clique and gate operations should match. In order to avoid putting gates of different operations into one clique, we assign operations to the cliques, such that the operation of the j-th clique equals the operation of the j-th gate. The gates are allowed to belong only to the cliques C_j of the same operation as the gate i is.

4. $g_j^j - g_i^j \geq 0$ for all $i \in G$, $j \in G$.
   If the clique C_j is non-empty, then it contains the gate indexed by j. This makes the gate j the representative of C_j.

5. $g_j^j = 1$ for all j such that $cost(\mathsf{op}^{\mathsf{G}}(j)) = 0$.
We are more interested in fusing the gates with positive cost. Actually, in some cases, even fusing gates of cost 0 can be useful, since it may in turn eliminate some *oc* gates. This constraint makes the optimization faster, although we may lose some valuable solution in this way.

6. (a) $\ell_i - \ell_k \geq 1$ for all $i, k \in G, \mathsf{pred}^{\mathsf{G}}(i, k)$;
   (b) $\mathcal{L}(|G|, 0, \ell_i, \ell_j, g_i^j)$ for all $i, j \in G$;
   (c) $\mathcal{L}(|G|, 0, \ell_j, \ell_i, g_i^j)$ for all $i, j \in G$;
   (d) $\ell_i \geq 0, \ell_i \leq |G|$.

   After the gates are fused into cliques, their dependencies on each other are not allowed to form cycles. We assign a level ℓ_i to each gate i. If i is a predecessor of k, then $\ell_i < \ell_k$, but to avoid degenerate solutions to the ILP, we introduce some difference between the levels. If a gate i belongs to the clique C_j, then $\ell_i = \ell_j$. We may split the implication $g_i^j = 1 \implies \ell_i = \ell_j$ into two parts $g_i^j = 1 \implies (\ell_i - \ell_j) \geq 0$, $g_i^j = 1 \implies (\ell_j - \ell_i) \geq 0$, reducing them to the constraint $\mathcal{L}$. We take the maximal value for ℓ_i as $|G|$, since we need at most $|G|$ distinct levels, even if each gate is assigned a unique level.

   We would also like to take into account the conditional predecessors.

   (e) $d_j = (1 - g_j^j)$ for all $j \in G$;
   (f) $\mathcal{F}(1, \{d_j\} \cup \{g_i^j \mid i \in G, i \neq j\}, c_j)$ for all $j \in G$.

   These constraints fix the variable c_j so, that $c_j = 1$ iff the gate j is fused with some other gate. That is, either $d_j = 1$, implying $g_j^j = 0$ or that j belongs to some other clique, or $\sum_{i \in G, i \neq j} g_i^j \geq 1$, implying that there is some other gate belonging to g_j^j.

   (g) $\mathcal{L}(|G|, 1, \ell_i, \ell_k, c_i)$ for all $i, k \in G, \mathsf{cpred}^{\mathsf{G}}(i, k)$.

   The last constraint states that, if $c_i = 1$, i.e. the gate i is fused with some other gate, then $l_k - l_i \geq 1$, i.e. the gate i should be computed strictly before its conditional predecessor k.

Extended constraints. The basic constraints are sufficient to optimize the total cost of the gates, at the same time avoiding cycles. Since computing the boolean conditions may also produce some additional costs, we define some more variables with associated constraints that take them into account.

7. The ℓ-th argument of C_j requires a new oc gate iff the number of distinct ℓ-th inputs used by the gates $i \in C_j$ is at least 2. We want to define the variables $sc^j_\ell = |\mathsf{args}^{G'}(j)[\ell]| - 1$, and $s^j_\ell = 1$ iff $\mathsf{args}^{G'}(j)[\ell]$ is an oc gate, allowing to estimate whether a new oc should be introduced.

   (a) $\mathcal{F}(1, \{g^j_i \mid i \in G, k = \mathsf{args}^{\mathsf{G}}(i)[\ell]\}, fx^{jk}_\ell)$.
   These constraints define $fx^{jk}_\ell = 1$ iff $k \in \mathsf{args}^{\mathsf{G}}(C_j)[\ell]$;

   (b) $\mathcal{P}(1, fx^{jk}_\ell, g^i_k, e^{jk}_{i\ell})$ for all $k \in V(\mathsf{G})$.
   These constraints define $e^{jk}_{i\ell} = 1$ iff $k \in \mathsf{args}^{\mathsf{G}}(C_j)[\ell]$, and $k \in C_i$.

   (c) $\mathcal{F}(1, \{e^{jk}_{i\ell} \mid k \in V(\mathsf{G})\}, fg^{ji}_\ell)$ for all $i \in G$.
   These constraints define $fg^{ji}_\ell = 1$ iff $\exists k \in C_i : \; k \in \mathsf{args}^{\mathsf{G}}(C_j)[\ell]$.

   (d) $fg^{jk}_\ell = fx^{jk}_\ell$ for $k \in I(\mathsf{G})$.
   Together with (7c), it defines $fg^{jk}_\ell = 1$ iff $C_k \in \mathsf{args}^{\mathsf{G}}(C_j)[\ell]$ for $k \in V(\mathsf{G})$, where for $k \in I(\mathsf{G})$ we denote $C_k = k$.

   (e) $\mathcal{F}(2, \{fg^{jk}_\ell \mid k \in V(\mathsf{G})\}, s^j_\ell)$.
   These constraints define $s^j_\ell = 1$ iff the total number of ℓ-th inputs after fusing the gates of C_j is at least 2.

   (f) $sc^j_\ell = \sum_{k \in V(\mathsf{G})} fg^{jk}_\ell - g^j_j$.
   These constraints define $sc^j_\ell = |\mathsf{args}^{\mathsf{G}}(C_j)[\ell]| - 1$ for a non-empty clique. If $g^j_j = 0$, then also $\sum_{k \in V(\mathsf{G})} fg^{jk}_\ell = 0$, so it is not counted for empty cliques. We discuss it in more details when we prove the feasibility of the task in Section 6.5.6.

8. Similarly, for the old oc gates, we are going to define $uc^j = |\mathsf{args}^{G'}(j)| - g^j_j$, and $u^j = 1$ iff $|\mathsf{args}^{G'}(j)| > 1$. The oc gate g_j will remain in G' iff the number of remaining distinct choices made by g_j is at least 2.

   (a) $\mathcal{F}(1, \{fg^{jk}_\ell \mid \ell \in [\mathsf{arity}^{\mathsf{G}}(j)] \cap 2\mathbb{N}\}, fg^{jk})$ for all $j \in G$, $k \in V(\mathsf{G})$, $\mathsf{op}^{\mathsf{G}}(j) = oc$.
   These constraints define $fg^{jk} = 1$ iff C_k is a choice of the oc gate j.

   (b) $\mathcal{F}(2, \{fg^{jk} \mid k \in V(\mathsf{G})\}, u^j)$ for all $j \in G$, $\mathsf{op}^{\mathsf{G}}(j) = oc$.
   These constraints define $u_j = 1$ iff there are at least 2 choices left for the oc gate j.

   (c) $uc^j_\ell = \sum_{k \in V(\mathsf{G})} fg^{jk} - g^j_j$ for all $j \in G$, $\mathsf{op}^{\mathsf{G}}(j) = oc$.
   These constraints define $uc^j_\ell = |\mathsf{args}^{G'}(j)| - g^j_j$ for $\mathsf{op}^{\mathsf{G}}(j) = oc$.

9. We would like to check whether the weakest precondition $\phi^{\mathsf{G}}(j)$ of the gate g_j must be computed. We want to define $b_j = 1$ iff $\phi^{\mathsf{G}}(j)$ is needed.

   (a) $\mathcal{F}(1, \{s^j_\ell \mid \ell \in [\mathsf{arity}^{\mathsf{G}}(j)]\}, t^j)$ for $j \in G$.
   This checks if there will be an oblivious choice of at least one input of the clique C_j. If it is so, then we will need to compute $\phi^{\mathsf{G}}(i)$ for $i \in C_j$.

   (b) $\mathcal{P}(1, g^j_i, t^j, t^j_i)$ for $j \neq i \in G$.

   (c) $t^j_j = 0$ for $j \in G$.
   The variable t^j_i now denotes if the weakest precondition of g_i is needed for the clique C_j. Since we may set one of the choices to negation of all the other choices, we may eliminate one of the weakest preconditions participating in the choice. We choose it to be the choice of gate j, and hence we set $t^j_j = 0$.

   (d) $b_i = \sum_{j \in G} t^j_i$ for $i \in G$.
   Since we know that each gate belongs to exactly one clique, we know that, for a fixed i, we have $t^j_i = 1$ for exactly one j, and so it suffices to sum them up.

Binary Constraints

Since we are dealing with a mixed integer program, we need to state explicitly that some variables are binary:

$$g^j_i \in \{0, 1\} \text{ for all } i, j \in G \ .$$

The statement $sc^j_\ell, uc^j_\ell \in \mathbb{Z}$, and the binariness of all the other variables (except $\ell_j \in \mathbb{R}$) follow from the binariness of g^j_i. We prove it in Section 6.5.6. We will need this property in the proofs of transformation correctness.

Feasibility

We want to be sure that the obtained integer linear program indeed has at least one solution.

Theorem 6.1. *For any $(G, X, Y) \in \mathcal{G}$, if $(A, \vec{b}, \vec{c}, \mathcal{I}) = T^{\rightarrow}_{ILP}(G, X, Y)$, then the integer linear programming task*

$$\textit{minimize } \langle \vec{c}, \vec{x} \rangle, \ \textit{s.t } A\vec{x} \leq \vec{b}, \vec{x} \geq \vec{0}, x_i \in \{0, 1\} \textit{ for } i \in \mathcal{I}$$

has at least one feasible solution.

The proof on this theorem can be found in Section 6.5.6. It just shows that it is always possible to take the solution where no gates are fused at all.

6.4.7 Circuit Transformation

Let $(G, X, Y) \in \mathcal{G}$ be the initial circuit. Let $Sol(G, X, Y)$ be the set of all feasible solutions to $(A, \vec{b}, \vec{c}, \mathcal{I}) = T^{\rightarrow}_{ILP}(G, X, Y)$. Now we define a backwards transformation $T^{\leftarrow}_{ILP} : Sol(\mathcal{G}) \times \mathcal{G} \rightarrow \mathcal{G}$, which takes any feasible solution to $(A, \vec{b}, \vec{c}, \mathcal{I})$ and applies it to (G, X, Y), forming a new circuit $\mathsf{G}' = (G', X', Y')$. Let $cost : \mathcal{G} \rightarrow \mathbb{Z}$ be function computing the total communication cost of a circuit.

The work of $T^{\leftarrow}_{ILP}$ is pretty straightforward. If fuses the gates according to the variables g^j_i of the ILP solution that denote which gate belongs to which clique. It introduces all the necessary oblivious choices, and also removes the old oblivious choices that are left with only one choice. The work of the transformation function $T^{\leftarrow}_{ILP}$ is given in Algorithm 7. Let sol be a mapping from ILP variables to their valuations. After evaluating sol by solving the ILP problem on line 1, the circuit is constructed sequentially, starting from an empty set of gates initialized on line 2.

The loop of line 3 iterates through all the cliques C_j. We assume that the cliques are sorted topologically, according to their level ℓ_j, so that the arguments of the clique are processed before the clique itself. The clique C_j is defined on line 4 as the set of all gates belonging to it. The arguments of C_j are processed one by one by the loop on line 5. On line 6 all the ℓ-th arguments of C_j and their weakest preconditions are collected into the set $\mathcal{B}^j_\ell$. Since we have computed the weakest preconditions ϕ^{G}_i in the initial graph G, some variables of ϕ^{G}_i may be unavailable in G' due to gate fusing. Hence each gate of ϕ^{G}_i is substituted with the corresponding clique representative that is left in G' after the fusing.

A fresh name v^j_ℓ is created for the new oc gate on line 7, where $\mathsf{fresh}()$ just creates a new variable name that has not been used anywhere else. Then, Algorithm 8 is called on line 8, and it actually decides if an oc gate is needed. On line 1 of Algorithm 8, all the values from which to choose are collected into the set $\mathcal{K}$. If $|\mathcal{K}| > 1$, then there are at least 2 choice candidates, and hence an oblivious choice needs to be introduced. The new node vb_k is needed to construct the choice of the argument k, which may be chosen by several different mutually exclusive choices. Hence the condition of choosing k is the sum of all the conditions b such that $(b, k) \in \mathcal{B}$ (here we are allowed to use addition instead of $\vee$ since the gates are mutually exclusive). The restriction T_C of T_P on private conditionals (defined formally in Section 6.5.2) transforms the boolean expression to a set of gates. The oc gate itself is formally constructed on line 6 of Algorithm 8. If $|\mathcal{K}| \leq 1$, then the new oc gate is not needed, and the the only element of $|\mathcal{K}|$ can be used straightforwardly. In the latter case, we substitute oc with an identity gate id to make the presentation simpler.

After the inputs of C_j are handled, if $\mathsf{op}^{\mathsf{G}}(j) \neq oc$, the representative of C_j is included into G on line 15. If $\mathsf{op}^{\mathsf{G}}(j) = oc$, the algorithm collects the choices and their conditions directly from the arguments of j, and calls Algorithm 8 to check

if it remains an oc gate, or becomes an id gate. This happens on lines 10-13 of Algorithm 7.

Some variables of Y may point to improper output wires if the corresponding gates have been fused into cliques. These references are rearranged on line 16.

Theorem 6.2. *Let* $(G, X, Y) \in \mathcal{G}$. *Let* solve *be an arbitrary integer linear programming solving algorithm. Let* $(A, \vec{b}, \vec{c}, \mathcal{I}) = T^{\rightarrow}_{ILP}(G, X, Y)$. *Then for any* $s \in State$, $[\![\texttt{eval}(G, X, Y)]\!]\, s = [\![\texttt{eval}(T^{\leftarrow}_{ILP}(\mathsf{solve}(A, \vec{b}, \vec{c}, \mathcal{I}), (G, X, Y)))]\!]\, s$.

Theorem 6.2 states the correctness of $T^{\rightarrow}_{ILP}$ and $T^{\leftarrow}_{ILP}$, i.e. that the semantics of the transformed circuit does not change. The proof of Theorem 6.2 is given in Section 6.5.7.

We may use Algorithm 7 to construct a circuit from a set of cliques obtained from some greedy algorithm of Section 6.4.5, as it can be easily reduced to a linear programming solution of $T^{\rightarrow}_{ILP}(G, X, Y)$.

Theorem 6.3. *Let* $(G, X, Y) \in \mathcal{G}$. *Let* greed *be the function of Algorithm 2 that returns a set of cliques of* G. *There exists a transformation* T^{C}_{G} *such that, for any* $s \in State$, $[\![\texttt{eval}(G, X, Y)]\!]\, s = [\![\texttt{eval}(T^{\leftarrow}_{ILP}(T^{C}_{G}(\mathsf{greed}(G), X, Y)))]\!]\, s$.

The proof of Theorem 6.3 is given in Section 6.5.7.

We want to estimate the communication cost of the obtained transformed graph (G', X', Y'). We show that its cost is the value estimated by the ILP, so its minimization is reasonable.

Theorem 6.4. *Let* $(G, X, Y) \in \mathcal{G}$. *Let* solve *be an arbitrary integer linear programming solving algorithm. Let* $(A, \vec{b}, \vec{c}, \mathcal{I}) = T^{\rightarrow}_{ILP}(G, X, Y)$. *Then* $cost(T^{\leftarrow}_{ILP}(\mathsf{solve}(A, \vec{b}, \vec{c}, \mathcal{I}), (G, X, Y))) = \langle \vec{c}, \mathsf{solve}(A, \vec{b}, \vec{c}) \rangle$.

The proof of Theorem 6.3 is given in Section 6.5.7.

If we use the greedy algorithms of Section 6.4.5, or use only the basic constrains of the integer program for better convergence, then it may happen that the found solution is worse than the initial one. Indeed, although the total cost of the gates may only decrease, the additional oblivious choices provide computational overhead that is not taken into account by these algorithms. If the oblivious choice is relatively expensive compared to the cost of fused gates, then the new oc gates may provide even larger overhead that the cost of the eliminated gates was.

From our benchmarks reported in Section 6.6, we see that the greedy algorithms nevertheless provide reasonable execution times, and their optimization times are significantly better in practice. The reason is that the oblivious choice is relatively cheap for the particular SMC platform that we use, and the gates that are involved into the optimization are significantly more expensive. Therefore, it is safe to omit the oblivious choices from the cost estimation.

Algorithm 7: $T^{\leftarrow}_{ILP}$ reconstructs the circuit G according to the variables g^j_i

Data: A circuit $\mathsf{G} = (G, X, Y) \in \mathcal{G}$
Data: An ILP $(A, \vec{b}, \vec{c})$
Result: A transformed circuit $\mathsf{G}' = (G', X', Y')$

$\mathsf{sol} \leftarrow \mathsf{solve}(A, \vec{b}, \vec{c})$;
$G' \leftarrow \emptyset$, $X' \leftarrow X$, $Y' \leftarrow Y$;
foreach $j \in G$, $\mathsf{sol}(g^j_j) = 1$ **do**
  $C_j \leftarrow \{i \mid \mathsf{sol}(g^j_i) = 1\}$;
  foreach $\ell \in [\mathsf{arity}^{\mathsf{G}}(j)]$ **do**
    $\mathcal{B}^j_\ell \leftarrow \{(\phi^{\mathsf{G}}_i[i' \leftarrow j' \mid i' \in C_{j'}], k) \mid k \in I(\mathsf{G}), i \in C_j, k = \mathsf{args}^{\mathsf{G}}(i)[\ell]\}$
      $\cup \{(\phi^{\mathsf{G}}_i[i' \leftarrow j' \mid i' \in C_{j'}], k) \mid k \notin I(\mathsf{G}), i \in C_j, \exists u :\ u = \mathsf{args}^{\mathsf{G}}(i)[\ell], u \in C_k\}$;
    $v^j_\ell \leftarrow \mathsf{fresh}()$;
    $G^j_\ell \leftarrow \mathsf{ocSubgraph}(v^j_\ell, \mathcal{B}^j_\ell)$;
    $G' \leftarrow G' \cup G^j_\ell$;
  if $\mathsf{op}^{\mathsf{G}}(j) = oc$ **then**
    $\mathcal{B}^j \leftarrow \{(\mathsf{args}^{\mathsf{G}}(j)[\ell - 1][i' \leftarrow j' \mid i' \in C_{j'}], v^j_\ell) \mid \ell \in \mathsf{arity}^{\mathsf{G}}(j), \ell \in 2\mathbb{N}\}$;
    $G^j \leftarrow \mathsf{ocSubgraph}(j, \mathcal{B}^j)$;
    $G' \leftarrow G' \cup G^j$;
  else
    $G' \leftarrow G' \cup \{(j, \mathsf{op}^{\mathsf{G}}(j), [v^j_1, \ldots, v^j_{\mathsf{arity}^{\mathsf{G}}(j)}])\}$;
$Y' \leftarrow Y'[i \leftarrow j \mid i \in C_j]$;
return (G', X', Y');

Algorithm 8: ocSubgraph constructs either an oc gate, or an id gate

Data: $\mathcal{B}$ – a set of condition and choice pairs (b, k)
Data: v – the name of the wire that outputs the choice result
Result: A set of gates computing the oc and its conditions

$\mathcal{K} \leftarrow \{k \mid \exists b : \ (b, k) \in \mathcal{B}\}$;
if $|\mathcal{K}| > 1$ **then**
  foreach $k \in \mathcal{K}$ **do**
    $vb_k \leftarrow \mathsf{fresh}()$;
    $(G_k, X_k, Y_k) \leftarrow T_C(vb_k := \sum_{(b,k)\in\mathcal{B}} b)$;
  return $\{G_k\}_{k\in\mathcal{K}} \cup \{(v, oc, [vb_k, k]_{k\in\mathcal{K}})\}$;
else
  $\{w\} \leftarrow \mathcal{K}$;
  return $\{(v, id, w)\}$;

6.5 Formal Constructions and Proofs

In this section we give some formal definitions and proofs that we have omitted before for better readability.

6.5.1 Circuit Composition

Let the circuit $\mathsf{G} = (G, X, Y)$ be defined as in Section 6.3.1. We define the composition of circuits as syntactic objects, and prove that the resulting circuit indeed computes the composition.

Lemma 6.1. *Let* $\mathsf{G}_1 = (G_1, X_1, Y_1)$ *and* $\mathsf{G}_2 = (G_2, X_2, Y_2)$ *where*

1. $V(\mathsf{G}_1) \cap V(\mathsf{G}_2) = \emptyset$*;*
2. $\mathsf{Dom}(Y_1) \cap \mathsf{Dom}(Y_2) = \emptyset$*;*
3. $\mathsf{Dom}(Y_1) \cap \mathsf{Ran}(X_2) = \emptyset$.

Defining a new circuit $\mathsf{G} = (G, X, Y)$ *where* $G := G_1 \cup G_2$*,* $X := X_1 \cup X_2$*,* $Y := Y_1 \cup Y_2$*, we get*

$$\forall s \in \mathit{State} : \ [\![\mathtt{eval}(G, X, Y)]\!]\, s = [\![\mathtt{eval}(\mathsf{G}_2)]\!]\, [\![\mathtt{eval}(\mathsf{G}_1)]\!]\, s \ .$$

Proof. Let us write out the definitions of expressions.

- $[\![\mathtt{eval}(G_1, X_1, Y_1)]\!]\, s = \mathtt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)$;

- $[\![\texttt{eval}(G_2, X_2, Y_2)]\!]\, s = \texttt{upd}(Y_2 \circ [\![G_2]\!](s \circ X_2), s)$;
- $[\![\texttt{eval}(G, X, Y)]\!]\, s = \texttt{upd}((Y_1 \cup Y_2) \circ [\![G_1 \cup G_2]\!](s \circ (X_1 \cup X_2)), s)$.

Let $y \in \mathit{Var}$ be any program variable. Let $s' = \texttt{upd}(Y \circ [\![G]\!](s \circ X), s)$, and let $s'' := \texttt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)$. We do the proof by case distinction on y.

- If $y \notin \mathsf{Dom}(Y_1) \cup \mathsf{Dom}(Y_2)$, then

$$\begin{aligned} s'(y) &= \texttt{upd}((Y_1 \cup Y_2) \circ [\![G_1 \cup G_2]\!](s \circ (X_1 \cup X_2)), s)\, y \\ &= s(y)\ . \end{aligned}$$

  On the other hand, we have

$$\begin{aligned} ([\![\texttt{eval}(\mathsf{G}_2)]\!]\, [\![\texttt{eval}(\mathsf{G}_1)]\!]\, s)\, y &= \texttt{upd}(Y_2 \circ [\![G_2]\!](s'' \circ X_2), s'')\, y \\ &= s''(y) \\ &= \texttt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)\, y \\ &= s(y)\ . \end{aligned}$$

- If $y \in \mathsf{Dom}(Y_i)$, then there exists $u \in V(\mathsf{G}_i)$ such that $Y_i(y) = u$. For any input wire valuations $W_1 : I(\mathsf{G}_1) \to \mathit{Val}$, $W_2 : I(\mathsf{G}_2) \to \mathit{Val}$, and since $V(\mathsf{G}_1) \cap V(\mathsf{G}_2) = \emptyset$, we can define $W_1 \cup W_2 = W : I(\mathsf{G}) \to \mathit{Val}$. We have

$$[\![G_1 \cup G_2]\!]\, W\, u = \begin{cases} [\![G_1]\!]\, W_1\, u \text{ if } u \in V(\mathsf{G}_1) \\ [\![G_2]\!]\, W_2\, u \text{ if } u \in V(\mathsf{G}_2) \end{cases}\ .$$

  1. Let $y \in \mathsf{Dom}(Y_1) \setminus \mathsf{Dom}(Y_2)$. Then

$$\begin{aligned} s'(y) &= \texttt{upd}(Y \circ [\![G]\!](s \circ X), s)\, y \\ &= \texttt{upd}((Y_1 \cup Y_2) \circ [\![G_1 \cup G_2]\!](s \circ (X_1 \cup X_2)), s)\, y \\ &= \texttt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)\, y \\ &= s''(y)\ . \end{aligned}$$

     Since updating the variables of $\mathsf{Dom}(Y_2)$ does not affect the value of $y \notin \mathsf{Dom}(Y_2)$, we have

$$\begin{aligned} s''(y) &= \texttt{upd}(Y_2 \circ [\![G_2]\!](s'' \circ X_2), s'')\, y \\ &= ([\![\texttt{eval}(\mathsf{G}_2)]\!]\, [\![\texttt{eval}(\mathsf{G}_1)]\!]\, s)\, y\ . \end{aligned}$$

  2. Let $y \in \mathsf{Dom}(Y_2)$. Then

$$\begin{aligned} s'(y) &= \texttt{upd}(Y \circ [\![G]\!](s \circ X), s)\, y \\ &= \texttt{upd}(Y_2 \circ [\![G_2]\!](s \circ X_2), s)\, y\ . \end{aligned}$$

Note that, for all $x \in \mathsf{Ran}(X_2)$, we have $s(x) = s''(x)$ due to the condition $\mathsf{Dom}(Y_1) \cap \mathsf{Ran}(X_2) = \emptyset$. We get

$$s'(y) = \mathtt{upd}(Y_2 \circ [\![G_2]\!](s'' \circ X_2), s)\, y \ .$$

Also, if $y \in \mathsf{Dom}(Y_2)$, then $s(y) = s''(y)$ due to the condition $\mathsf{Dom}(Y_1) \cap \mathsf{Dom}(Y_2) = \emptyset$. We get

$$\begin{aligned} s'(y) &= \mathtt{upd}(Y_2 \circ [\![G_2]\!](s'' \circ X_2), s'')\, y \\ &= ([\![\mathtt{eval}(\mathsf{G}_2)]\!]\, [\![\mathtt{eval}(\mathsf{G}_1)]\!]\, s)\, y \ . \end{aligned}$$

Hence, the claim is proven for all $y \in \mathit{Var}$. □

6.5.2 Transformations of Programs to Circuits

We define a transformation $T_P : \mathit{prog} \to \mathit{prog}$ that substitutes all private conditionals of the initial program with circuit evaluations. The transformation T_P does not modify statements outside of the private conditionals. If it is applied to a private conditional, it uses an auxiliary transformation $T_C : \mathit{statement} \to \mathcal{G}$ to construct a circuit, and substitutes the private conditional block with $\mathtt{eval}(G, X, Y)$, where the circuit (G, X, Y) is generated by T_C. We give the recursive definitions of T_P and T_C.

- $T_P(S_1 \ ; \ S_2) = T_P(S_1) \ ; \ T_P(S_2)$.
- $T_P(a := b) = (a := b)$.
- $T_P(\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2) = \mathtt{if}\ b\ \mathtt{then}\ T_P(S_1)\ \mathtt{else}\ T_P(S_2)$ for a public b.
- $T_P(\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2) = \mathtt{eval}(G, X, Y)$ where $(G, X, Y) = T_C(\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2)$ for a private b.

The transformation T_C creates the circuits corresponding to the computation inside the private conditionals, and arranges the mappings X and Y that establish relations between the circuit wires and the program variables.

- $T_C(\mathtt{skip}) = (\emptyset, \emptyset, \emptyset)$.
- $T_C(y := x) = (\emptyset, \{v \leftarrow x\}, \{y \leftarrow v\})$, where x is a variable name or a constant. There are no gates, and the value for y is taken directly from the wire to which the value of x is assigned.
- $T_C(y := f(x_1, \ldots, x_n)) = (G, X, Y)$, where

- $x_1, \ldots, x_n$ are program variables and constants;
- f is some arithmetic blackbox function defined in the programming language;
- $G = (w, [\![f]\!], [v_1, \ldots, v_n])$ is a gate that computes f;
- $X = \{v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n\}$;
- $Y = \{y \leftarrow w\}$.

- $T_C(y := f(e_1, \ldots, e_n))$
  $= T_C(y_{i_1} := e_{i_1} \ ; \ \ldots \ ; \ y_{i_n} := e_{i_n}; \ y := f(y_1, \ldots, y_n))$, where
  - $\{e_{i_1}, \ldots, e_{i_n}\} \subseteq \{e_1, \ldots, e_n\}$ are compound expressions (not variables/constants);
  - $y_i = e_i$ for $i \notin \{i_1, \ldots, i_n\}$.

- $T_C(S_1 \ ; \ S_2) = (G, X, Y)$ where
  - $(G_i, X_i, Y_i) = T_C(S_i)$ for $i \in \{1, 2\}$;
  - $X = X_1 \cup X_2'$ where $X_2' = (X_2 \setminus \{v \leftarrow x \mid x \in \mathsf{Dom}(Y_1)\})$: the inputs of both X_1 and X_2 will be needed during the computation, but the variables of X_2 that are modified by S_1 should be taken from the output of S_1's circuit instead.
  - $Y = Y_2 \cup Y_1'$ where $Y_1' = (Y_1 \setminus \{y \leftarrow w \mid y \in \mathsf{Dom}(Y_2)\})$: all the variables that are modified throughout the execution of $S_1 \ ; \ S_2$ are in Y. If a variable is modified in both S_1 and S_2, its value is taken from the output of S_2.
  - Now G_1 and G_2 should be combined. Take $G = G_1 \cup G_2'$ where $G_2' = G_2[\{v \leftarrow w \mid (v \leftarrow x) \in X_2, (x \leftarrow w) \in Y_1\}]$. This connects the inputs of G_2 to the outputs of G_1.

- $T_C(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = T_C(S)$ where
  - $(G_i, X_i, Y_i) = T_C(S_i)$ for $i \in \{1, 2\}$,
  - $Y_i' = \{z_{iy} \leftarrow w \mid (y \leftarrow w) \in Y_i\}$ for $i \in \{1, 2\}$: this renames the variables y of Y_i by introducing new variable names z_{iy},
  - $$S = \begin{cases} b_1 := b; \\ b_2 := (1 - b); \\ S_1' = S_1[(y \leftarrow z_{1y}) \in Y_1' \mid \exists w \ (y \leftarrow w) \in Y_1]; \\ S_2' = S_2[(y \leftarrow z_{2y}) \in Y_2' \mid \exists w \ (y \leftarrow w) \in Y_2]; \\ y := oc(b_1, r(1, y), b_2, r(2, y)) \ \forall y \in Y; \end{cases},$$

– $r(i,y) = \begin{cases} Y_i'(w) \text{ if } (y \leftarrow w) \in Y_i \\ y \text{ otherwise} \end{cases}$.

This computes S_1 and S_2 sequentially (renaming all the assigned variables in order to ensure that there are no conflicts), and then applies a new binary oblivious choice (b or $\neg b$) to the outputs of G_1 and G_2. All the outputs of all the branches should be available in an oblivious selection. If any variables are modified in only one of the branches, they should be output also by the circuit that corresponds to the other branch, in order to make them indistinguishable. Such variables y are just copied from the input directly.

As the result, if P is the initial program with private conditions, $T_P(P)$ is a program without private conditions, but with some instances of the function call $\texttt{eval}(G, X, Y)$ in its code. We need to prove that $T_P(P)$ does the same computation as P.

Theorem 6.5. *For any program P, $s \in State$, $[\![T_P(P)]\!]\, s = [\![P]\!]\, s$.*

Proof. It is sufficient to prove the correctness of T_P, which in turn will require us to prove the correctness of T_C. Since the transformations T_P and T_C are defined inductively, we prove their correctness inductively on the size of P. Let $[\![T_P(S_1)]\!] = [\![S_1]\!]$, $[\![T_P(S_2)]\!] = [\![S_2]\!]$.

- Using the definition $T_P(S_1\ ;\ S_2) = T_P(S_1)\ ;\ T_P(S_2)$, we get

$$\begin{aligned} [\![T_P(S_1\ ;\ S_2)]\!]\, s &= [\![T_P(S_1)\ ;\ T_P(S_2)]\!]\, s \\ &= [\![T_P(S_2)]\!]\, [\![T_P(S_1)]\!]\, s \\ &= [\![S_2]\!]\, [\![S_1]\!]\, s = [\![(S_1\ ;\ S_2)]\!]\, s \ . \end{aligned}$$

- Using the definition $T_P(a := b) = (a := b)$, we get

$$[\![T_P(a := b)]\!]\, s = [\![(a := b)]\!]\, s \ .$$

- Using $T_P(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = \texttt{if}\ b\ \texttt{then}\ T_P(S_1)\ \texttt{else}\ T_P(S_2)$ for a public b, we get

$$\begin{aligned} S &= [\![\texttt{if}\ b\ \texttt{then}\ T_P(S_1)\ \texttt{else}\ T_P(S_2)]\!]\, s \\ &= \begin{cases} [\![T_P(S_1)]\!]\, s \text{ if } [\![b]\!]\, s \neq 0 \\ [\![T_P(S_2)]\!]\, s \text{ if } [\![b]\!]\, s = 0 \end{cases} \\ &= \begin{cases} [\![S_1]\!]\, s \text{ if } [\![b]\!]\, s \neq 0 \\ [\![S_2]\!]\, s \text{ if } [\![b]\!]\, s = 0 \end{cases} \\ &= [\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\, s \ . \end{aligned}$$

- Using the definition $T_P(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = \texttt{eval}(G, X, Y)$ for a private b, where $(G, X, Y) = T_C(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2)$, assuming the correctness of T_C, we get

$$[\![\texttt{eval}(G, X, Y)]\!]\, s = [\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\, s \ .$$

We now prove the correctness of T_C. By definition of T_C, we need to prove $[\![\texttt{eval}(G, X, Y)]\!] = [\![P]\!]$ for $(G, X, Y) = T_C(P)$.

- Using the definition $T_C(skip) = (\emptyset, \emptyset, \emptyset)$, we get

$$[\![\texttt{eval}(\emptyset, \emptyset, \emptyset)]\!]\, s = \texttt{upd}(\emptyset, s) = s = [\![skip]\!]\, s \ .$$

- Using the definition $T_C(y := x) = (\emptyset, \{v \leftarrow x\}, \{y \leftarrow v\})$, where x is a variable name or a constant, we get

  $[\![\texttt{eval}(\emptyset, \{v \leftarrow x\}, \{y \leftarrow v\})]\!]\, s$

$$\begin{aligned} &= \texttt{upd}((y \leftarrow v) \circ [\![\emptyset]\!](s \circ (v \leftarrow x)), s) \\ &= \texttt{upd}((y \leftarrow v) \circ s \circ (v \leftarrow x), s) \\ &= s[y \leftarrow s(x)] \\ &= [\![y := x]\!]\, s \ . \end{aligned}$$

- Using the definition $T_C(y := f(x_1, \ldots, x_n)) = (G, X, Y)$, where
  - $x_1, \ldots, x_n$ are program variables and constants;
  - f is some arithmetic blackbox function defined in the programming language;
  - $G = (w, [\![f]\!], [v_1, \ldots, v_n])$ is a gate that computes f;
  - $X = \{v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n\}$;
  - $Y = \{y \leftarrow w\}$;

  we get

  $[\![\texttt{eval}(G, \{v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n\}, \{y \leftarrow w\})]\!]\, s$

$$\begin{aligned} &= \texttt{upd}((y \leftarrow w) \circ [\![G]\!](s \circ \{v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n\})), s) \\ &= \texttt{upd}((y \leftarrow w) \circ s'[w \leftarrow f(x_1, \ldots, x_n)], s) \\ &= \texttt{upd}((y \leftarrow f(x_1, \ldots, x_n)), s) \\ &= s[y \leftarrow f(x_1, \ldots, x_n)] \\ &= [\![(y := f(x_1, \ldots, x_n)]\!]\, s \ . \end{aligned}$$

- Let us use the definition $T_C(S_1\ ;\ S_2) = (G, X, Y)$ where
  - $(G_i, X_i, Y_i) = T_C(S_i)$ for $i \in \{1, 2\}$;
  - $X = X_1 \cup X_2'$ for $X_2' = (X_2 \setminus \{v \leftarrow x \mid x \in \mathsf{Dom}(Y_1)\})$;
  - $Y = Y_2 \cup Y_1'$ for $Y_1' = (Y_1 \setminus \{y \leftarrow w \mid y \in \mathsf{Dom}(Y_2)\})$;
  - $G = G_1 \cup G_2'$ for $G_2' = G_2[\{v \leftarrow w \mid (v \leftarrow x) \in X_2, (x \leftarrow w) \in Y_1\}]$.

  Define the following auxiliary subsets:

$$\begin{aligned} {X_2}^{Y_1} &= \{(v \leftarrow x) \in X_2 \mid x \in \mathsf{Dom}(Y_1)\}\ , \\ {Y_2}^{Y_1} &= \{(y \leftarrow w) \in Y_2 \mid y \in \mathsf{Dom}(Y_1)\}\ . \end{aligned}$$

  Since the wires that are not evaluated by X_2' are defined inside G_2 as $G_2[{X_2}^{Y_1}]$, we may write

$$\begin{aligned} [\![\mathtt{eval}(G_2, X_2, Y_2)]\!]\, s &= [\![\mathtt{eval}(G_2[{X_2}^{Y_1}], X_2 \setminus {X_2}^{Y_1}, Y_2)]\!]\, s \\ &= [\![\mathtt{eval}(G_2[{X_2}^{Y_1}], X_2', Y_2)]\!]\, s\ . \end{aligned}$$

  Note that

$$\begin{aligned} {X_2}^{Y_1} \circ Y_1 &= \{(v \leftarrow x) \in X_2 \mid x \in \mathsf{Dom}(Y_1)\} \circ \{(y \leftarrow w) \in Y_1\} \\ &= \{v \leftarrow w \mid (v \leftarrow x) \in X_2, (x \leftarrow w) \in Y_1\}\ . \end{aligned}$$

  Hence we have $G_2' = G_2[{X_2}^{Y_1} \circ Y_1]$.

  We may now write out the composition:

$$\begin{aligned} &[\![\mathtt{eval}(G_2, X_2, Y_2)]\!]\, [\![\mathtt{eval}(G_1, X_1, Y_1)]\!]\, s \\ &= [\![\mathtt{eval}(G_2[{X_2}^{Y_1}], X_2 \setminus {X_2}^{Y_1}, Y_2)]\!]\, [\![\mathtt{eval}(G_1, X_1, Y_1)]\!]\, s \\ &= [\![\mathtt{eval}(G_2[{X_2}^{Y_1} \circ Y_1], X_2 \setminus {X_2}^{Y_1}, Y_2)]\!]\, [\![\mathtt{eval}(G_1, X_1, Y_1 \setminus {Y_2}^{Y_1})]\!]\, s \\ &= [\![\mathtt{eval}(G_2', X_2', Y_2)]\!]\, [\![\mathtt{eval}(G_1, X_1, Y_1')]\!]\, s\ . \end{aligned}$$

  By definition of Y_1' and X_2', we have $\mathsf{Dom}(Y_1') \cap \mathsf{Dom}(Y_2) = \emptyset$ and $\mathsf{Dom}(Y_1) \cap \mathsf{Ran}(X_2') = \emptyset$. Applying Lemma 6.1, we get

$$\begin{aligned} &[\![\mathtt{eval}(G_2', X_2', Y_2)]\!]\, [\![\mathtt{eval}(G_1, X_1, Y_1')]\!]\, s \\ &\qquad = [\![\mathtt{eval}(G_1 \cup G_2', X_1 \cup X_2', Y_1' \cup Y_2)]\!]\, s. \end{aligned}$$

  Putting together these equations, by definition of G, X, Y, we get

$$\begin{aligned} &[\![\mathtt{eval}(G_2, X_2, Y_2)]\!]\, [\![\mathtt{eval}(G_1, X_1, Y_1)]\!]\, s \\ &\qquad = [\![\mathtt{eval}(G_1 \cup G_2', X_1 \cup X_2', Y_1' \cup Y_2)]\!] \\ &\qquad = [\![\mathtt{eval}(G, X, Y)]\!]\ . \end{aligned}$$

  By the induction hypothesis, $[\![\mathtt{eval}(G_i, X_i, Y_i)]\!]\, s = [\![S_i]\!]\, s$ for $i \in \{1, 2\}$. Hence we have $[\![\mathtt{eval}(G, X, Y)]\!] = [\![S_2]\!]\, [\![S_1]\!]\, s = [\![S_1; S_2]\!]\, s$.

- Use the definition $T_C(y := f(e_1, \ldots, e_n)) = T_C(y_{i_1} := e_{i_1} \ ; \ \ldots \ ; \ y_{i_n} := e_{i_n}; \ y := f(y_1, \ldots, y_n))$, where $e_{i_1}, \ldots, e_{i_n}$ are compound expressions (i.e neither variables nor constants), and $y_i = e_i$ for $i \notin \{i_1, \ldots, i_n\}$.

  Since we have already defined T_C on a sequential composition, by the induction hypothesis, $T_C(y_{i_1} := e_{i_1} \ ; \ \ldots \ ; \ y_{i_n} := e_{i_n}) = (G_1, X_1, Y_1)$ such that, for each $i \in [n]$, we have $Y_1(y_i) = [\![e_i]\!]$. We also have $T_C(y := f(y_1, \ldots, y_n)) = (G_2, X_2, Y_2)$ where $Y_2(y) = [\![f(y_1, \ldots, y_n)]\!]$, as $y_1, \ldots, y_n$ are now all either program variables or constants, so it has also been treated on the previous induction steps. Composing them together, we get $T_C(y := f(e_1, \ldots, e_n)) = (G, X, Y)$ such that $Y(y) = [\![f]\!]([\![e_1]\!]s, \ldots, [\![e_n]\!]\, s)y = [\![f(e_1, \ldots, e_n)]\!]$.

- Let us use the definition $T_C(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = T_C(S)$ where

  - $(G_i, X_i, Y_i) = T_C(S_i)$ for $i \in \{1, 2\}$,
  - $Y_i' = \{z_{iy} \leftarrow w \mid (y \leftarrow w) \in Y_i\}$ for $i \in \{1, 2\}$: this renames the variables y of Y_i by introducing new variable names z_{iy},
  - $S = \begin{cases} b_1 := b; \\ b_2 := (1 - b); \\ S_1' = S_1[(y \leftarrow z_{1y}) \in Y_1' \mid \exists w \ (y \leftarrow w) \in Y_1]; \\ S_2' = S_2[(y \leftarrow z_{2y}) \in Y_2' \mid \exists w \ (y \leftarrow w) \in Y_2]; \\ y := oc(b_1, r(1, y), b_2, r(2, y)) \ \forall y \in Y; \end{cases}$,
  - $r(i, y) = \begin{cases} Y_i'(w) \text{ if } (y \leftarrow w) \in Y_i \\ y \text{ otherwise} \end{cases}$.

  First, we claim that $[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\, s = [\![S]\!]\, s$. Let $y \in \mathit{Var}$.

  - If $y \notin Y$, then $([\![S]\!]\, s)\, y = s(y)$ since S reassigns only b_1 and b_2, which are not the part of s, and $y \in Y$. At the same time, if $y \notin Y$, then $y \notin (Y_1 \cup Y_2)$, and since S_i reassigns only variables of Y_i, $([\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\, s)\, y = s(y)$.
  - Let $y \in Y$. Let $[\![b]\!]\, s = 1$, Then $b_1 = 1$, and $b_2 = 0$, so $y = oc((b_1, r(1, y)), (b_2, r(2, y)) = r(1, y)$.
    - If $(y \leftarrow w) \in Y_1$, then $r(1, y) = Y_1'(w) = z_{1y}$. The only place where z_{1y} can be assigned is statement S_1', and we have $([\![S_1[(y \leftarrow z_{1y}) \in Y_1' \mid (y \leftarrow w) \in Y_1]]\!]\, s)\, z_{1y} = ([\![S_1]\!]\, s)\, y$. Hence if $b = 1$, then $([\![S]\!]\, s)\, y = ([\![S_1]\!]\, s)\, y$.
    - Otherwise, $r(1, y) = y$, so the final statement is $y := y$. Since it is the first assignment of y in S, we have $([\![S]\!]\, s)\, y = s(y)$. At

the same time, $(\llbracket S_1 \rrbracket\, s)\, y = s(y)$ since if $z_{1y} \notin \mathsf{Dom}(Y_1')$, then $y \notin \mathsf{Dom}(Y_1)$, and hence it is not reassigned in S_1.

The proof is analogous for $\llbracket b \rrbracket\, s = 0$. We have got that:

$$\begin{aligned}(\llbracket S \rrbracket\, s)\, y &= \begin{cases}(\llbracket S_1 \rrbracket\, s)\, y \text{ if } \llbracket b \rrbracket\, s = 0\\ (\llbracket S_2 \rrbracket\, s)\, y \text{ otherwise}\end{cases}\\ &= (\llbracket \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \rrbracket\, s)\, y\ .\end{aligned}$$

By the induction hypothesis, $\llbracket \texttt{eval}(G_i, X_i, Y_i) \rrbracket\, s = \llbracket S_i \rrbracket\, s$ for $i \in \{1, 2\}$. Hence we have

$$\begin{aligned}\llbracket S_i' \rrbracket &= \llbracket S_i[(y \leftarrow z_{iy}) \in Y_i' \mid (y \leftarrow w) \in Y_i] \rrbracket\\ &= \llbracket \texttt{eval}(G_i, X_i, Y_i)[(y \leftarrow z_{iy}) \in Y_i' \mid (y \leftarrow w) \in Y_i] \rrbracket\\ &= \llbracket \texttt{eval}(G_i, X_i, Y_i') \rrbracket\ .\end{aligned}$$

Let us denote $S_C = (y := oc(b_1, r(1, y), b_2, r(2, y)))_{\forall y \in Y}$, and $S_b = (b_1 := b\ ;\ b_2 := (1 - b))$. We take $(G', X', Y') = T_C(S_b\ ;\ S_1'\ ; S_2'\ ;\ S_C)$. Assuming by the induction that T_C is defined correctly on sequential composition, we get

$$\llbracket \texttt{eval}(G', X', Y') \rrbracket\, s = \llbracket S_b\ ;\ S_1'\ ; S_2'\ ;\ S_C \rrbracket\, s\ . \qquad \square$$

6.5.3 Correctness of the WP Generating Algorithm

Proposition 6.1. On input $\mathsf{G} \in \mathcal{G}$, Algorithm 1 returns a mapping ϕ such that, for all $v \in V(\mathsf{G})$, $\phi^{\mathsf{G}}(v)$ is the weakest precondition of v according to Definition 6.5.

The proof of Algorithm 1 is split into two steps: the correctness of ψ definition and the correctness of ϕ^{G} definition. We prove it as two separate lemmata.

Lemma 6.2. *For each $v \in V(\mathsf{G})$ and for any ϕ_{in}, $\mathsf{process}(v, \phi_{in})$ returns a boolean expression ψ_{out} over $V(\mathsf{G})$ such that $\llbracket \psi_{out} \rrbracket\, \llbracket G \rrbracket(s \circ X) = \llbracket G \rrbracket(s \circ X)\, v$ (i.e. ψ_{out} is an expression over $V(\mathsf{G})$ that computes the same value as v).*

Proof. The proof is based on induction on the number of vertices v for which $\psi_{out}(v)$ has already been computed, starting from the inputs $I(\mathsf{G})$.

- **Base:** for $v \in I(\mathsf{G})$, the algorithm returns $\psi_{out} = v$ (there is no gate operation), so $\llbracket \psi_{out} \rrbracket\, \llbracket G \rrbracket(s \circ X) = \llbracket u := v \rrbracket\, \llbracket G \rrbracket(s \circ X)\, u = \llbracket G \rrbracket(s \circ X)[v]$.
- **Step:** If $\phi(v) \neq \bot$, then v has already been processed, and the algorithm returns $\psi_{out}(v)$, which is correct by the induction hypothesis. Let us now assume that $\phi(v) = \bot$. Let $\llbracket \psi_{out}^i \rrbracket\, \llbracket G \rrbracket(s \circ X) = \llbracket G \rrbracket(s \circ X)[v_i]$ for all $v_i \in \mathsf{args}^{\mathsf{G}}(v)$. There are now several cases for $\mathsf{op}^{\mathsf{G}}(v)$.

– If $\mathsf{op}^{\mathsf{G}}(v) = \wedge$, then

$$\begin{aligned}[\![G]\!](s \circ X)[v] &= [\![\wedge]\!]([\![G]\!](s \circ X)[v_1], [\![G]\!](s \circ X)[v_2]) \\ &= [\![\wedge]\!]([\![\psi^1_{out}]\!]\,[\![G]\!](s \circ X), [\![\psi^2_{out}]\!]\,[\![G]\!](s \circ X)) \\ &= [\![\psi_{out}]\!]\,[\![G]\!](s \circ X)\end{aligned}$$

The proof is analogous for $\vee$.

– If $\mathsf{op}^{\mathsf{G}}(v) = oc$, $\mathsf{arity}^{\mathsf{G}}(v) = n$, then

$$[\![\psi_{out}]\!]\,[\![G]\!](s \circ X) = \sum_{i=1}^{n}([\![G]\!](s \circ X)[b_i]) \cdot ([\![G]\!](s \circ X)[a_i]) \ ,$$

where $(a_i, b_i) \in \mathsf{args}^{\mathsf{G}}(v)$. By Definition 6.3, we have

$$[\![G]\!](s \circ X)[v] = \sum_{i=1}^{n}([\![G]\!](s \circ X)[b_i]) \cdot ([\![G]\!](s \circ X)[a_i]).$$

– Otherwise, the algorithm returns $\psi_{out} = v$. This is similar to the base case. □

Lemma 6.3. *Algorithm 1 outputs ϕ such that for all $v \in G$ and $s \in State$ we have $\phi(v) = 1$ iff $\mathsf{used}(v, s) = 1$ (according to Definition 6.4).*

Proof. The proof is based on induction, starting from the subset of outputs $O_f(\mathsf{G}) \subseteq O(\mathsf{G})$ that are *not used by any other gate as an argument*. Since our circuits are finite acyclic graphs, at least one such output does exist.

- **Base:** for $v \in O_f(\mathsf{G})$, the function process takes the argument $\phi_{in} = 1$. Since each $v \in O_f(\mathsf{G})$ is not an input of any other gate, it is visited only once. In this case, $\phi(v) = \bot$, and the algorithm assigns $\phi(v) = \phi_{in} = 1$. We have $[\![\phi(v)]\!]\, s = 1$ for all $s \in State$, and by the condition (1) of Definition 6.4, for all $v \in O(G)$ we have $\mathsf{used}(v, s) = 1$.

- **Step:** The definition of $\phi(v)$ may be constructed in several steps if v is visited several times. Only the final result needs to satisfy the lemma statement. Let $[\![\phi_{v_i}]\!]\, s = 1$ iff $\mathsf{used}(v_i, s) = 1$ for all v_i such that $v \in \mathsf{args}^{\mathsf{G}}(v_i)$. By Lines 7 and 10, we finally have $\phi(v) = \phi^1_{in} \vee \cdots \vee \phi^n_{in}$, where ϕ^i_{in} has been passed as the second argument of $\mathsf{process}(v, \phi^i_{in})$ by its successor v_i. The exact value of ϕ^i_{in} depends on $\mathsf{op}^{\mathsf{G}}(v_i)$.

  – If $\mathsf{op}^{\mathsf{G}}(v_i) \neq oc$, then $\mathsf{process}(v, \phi^i_{in})$ may be called on the Lines 14, 15, or 27. In all cases, $\phi^i_{in} = \phi(v_i)$, since ϕ^i_{in} was passed as not

a value, but as a reference, so it updates dynamically and is finally equal to $\phi(v_i)$. We get $\phi(v) = \phi(v_1) \vee \cdots \vee \phi(v_n)$. This is sufficient for the proof since $[\![\phi(v_i)]\!]\, s = 1$ iff $\mathsf{used}(v_i, s) = 1$, and having $\phi(v) = \phi(v_1) \vee \cdots \vee \phi(v_n)$ assigns $[\![\phi(v)]\!]\, s = 1$ if for at least one i we have $s(v_i) = 1$. This satisfies the condition (2) of Definition 6.4, since if $\mathsf{used}(v_i, s) = 1$, then $\mathsf{used}(v, s) = 1$.

– If $\mathsf{op}^{\mathsf{G}}(v) = oc$, then $\mathsf{process}(v, \phi^i_{in})$ for an odd i is called at the Line 20 with $\phi^i_{in} = \phi(v_i)$. For an even i, this happens on the Line 21 with $\phi^i_{in} = b'_i \wedge \phi(v_i)$ where $[\![b'_i]\!]\, [\![G]\!](s \circ X) = [\![G]\!](s \circ X)[b_i]$ by Lemma 6.2. Let s be now fixed. Let j be such that $[\![G]\!](s \circ X)[b_j] = 1$. For all even arguments, we have $[\![\phi^j_{in}]\!]\, s = [\![\phi(v_j)]\!]\, s$ and $[\![\phi^i_{in}]\!] s = 0$ for all $i \neq j$. This satisfies the condition (3) of Definition 6.4: if $\mathsf{used}(v_i, s) = 1$, then $\mathsf{used}(v, s) = 1$ if v is an odd argument b_i, or an even argument a_j such that $[\![G]\!](s \circ X)[b_j] = 1$.

So far we have proven that $[\![\phi^{\mathsf{G}}_v]\!]\, s = 1$ implies $\mathsf{used}(v, s)$. Now we prove the other direction. Let $v \in \mathsf{args}^{\mathsf{G}}(v_i)$. We have $[\![\phi^{\mathsf{G}}_v]\!]\, s = 1$ if there exists v_i such that $[\![\phi^{\mathsf{G}}_{v_i}]\!]\, s = 1$ and at least one of the following conditions holds:

– $\mathsf{op}^{\mathsf{G}}(v_i) \neq oc$;
– $\mathsf{op}^{\mathsf{G}}(v_i) = oc$, and v is an odd input of v_i;
– $\mathsf{op}^{\mathsf{G}}(v_i) = oc$, $[\![G]\!](s \circ X)[b_i] = 1$, and v is an even input of v_i.

Hence if conditions (1-3) of Definition 6.4 are satisfied, then $[\![\phi^{\mathsf{G}}_v]\!]\, s = 1$. □

Lemma 6.2 and Lemma 6.3 together immediately prove Proposition 6.1.

6.5.4 Correctness of the Subcircuit Partitioning Algorithm

Let the sets A_n be defined as in Section 6.4.4. We need to show that, after the gates are merged into subcircuits, the resulting circuit still has the same impact on the state as the initial circuit.

First, we need to formally define the operation that a subcircuit S computes, as we did for the gates. By construction, each A_k has only one output wire $w \in O(S)$, since except the root gate, we do not include any subgraphs whose outputs are used by some other subgraphs. Hence, for $v_i \in I(S)$, $n = |\mathsf{args}^{A_k}(S)|$, we may define the operation computed by S as

$$\mathsf{op}^{A_k}(S)(x_1, \ldots, x_n) = [\![S]\!]\, (v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n)\, w \ .$$

Theorem 6.6. *Let $G \in \mathcal{G}$, $n \in \mathbb{N}$. Then the following statements hold:*

- $I(\mathsf{G}) = I(A_n)$;

- *for all* $W : I(\mathsf{G}) \to \mathit{Val}$ *we have* $[\![G]\!](W) = [\![A_n]\!](W)$.

Proof. For shortness of notation, let us define the predicate $\mathsf{correct}(S)$ for $S \in A_n$, s.t $\mathsf{correct}(S) = 1$ iff for all $W : I(S) \to \mathit{Val}$, $w \in O(S)$ we have

$$[\![S]\!]\,([\![A_n]\!](W)\mathsf{args}^{A_n}(S))\,w = [\![S]\!]\,([\![G]\!](W)\mathsf{args}^{\mathsf{G}}(S))\,w \ .$$

In other words, $\mathsf{correct} = 1$ iff the output of S is the same, regardless of whether its inputs are evaluated in A_n or G. This property is a bit weaker than the one we are proving.

- **Base:** If $n = 0$, then each gate g is treated as a separate subcircuit, so $\mathsf{op}^{A_0}(\{g\}) = \mathsf{op}^{\mathsf{G}}(g)$, $\mathsf{args}^{A_0}(\{g\}) = \mathsf{args}^{\mathsf{G}}(g)$, $\mathsf{arity}^{A_0}(\{g\}) = \mathsf{arity}^{\mathsf{G}}(g)$. The circuit has not changed, so $G = A_0$, and hence $I(\mathsf{G}) = I(A_0)$, and $\forall W : I(\mathsf{G}) \to \mathit{Val}, [\![G]\!](W) = [\![A_0]\!](W)$. Moreover, $\mathsf{correct}(S)$ holds for any subcircuit S of A_0.

- **Step:** Assume that we have $I(\mathsf{G}) = I(A_n)$, and $\forall W : I(\mathsf{G}) \to \mathit{Val}$, $[\![G]\!](W) = [\![A_n]\!](W)$ for a circuit A_n obtained for depth n. Now we are trying to unite each subcircuit S of A_n with $\mathsf{args}^{A_n}(S)$. Only those elements of $\mathsf{args}^{\mathsf{G}}(S)$ that are used as arguments only by S are added, and each element is used on the current iteration only once. Hence, if the subcircuits of A_n are mutually exclusive, then so are the subcircuits of A'_{n+1}.

  The subcircuits S that occur at most one time are decomposed back to the subcircuits S' of A_n, and by the induction hypothesis $I(S') = I(T')$, $\forall W : I(S') \to \mathit{Val}, [\![S']\!](W) = [\![T']\!](W)$, where T' is a set of circuits of the gates of S' in G.

  The subcircuit S that occurred at least 2 times is left in A_{n+1}, and the mapping $\mathsf{args}^{A_{n+1}}(S)$ is updated. Each such subcircuit is of the form $S = \{S_0, S_1, \ldots, S_n\}$, where $\{S_1, \ldots, S_n\} \subseteq \mathsf{args}^{A_n}(S_0)$, and $\forall i \in \{0, \ldots, n\}$, $\mathsf{correct}(S_i)$ hold by the induction hypothesis. For all $W : I(S_i) \to \mathit{Val}$, we have

  $$[\![S_i]\!]([\![A_{n+1}]\!](W)\mathsf{args}^{A_n}(S_i))\,w_i = [\![S_i]\!]([\![G]\!](W)\mathsf{args}^{A_n}(S_i))\,w_i$$

  for $w_i \in O(S_i)$, and hence the subcircuits $\{S_1, \ldots, S_n\}$ provide to S_0 the same inputs it would get in G. Therefore S_0 also outputs the same value it would output in G, so for all $W : I(S) \to \mathit{Val}$ we have

  $$[\![S]\!]([\![A_{n+1}]\!](W)\mathsf{args}^{A_{n+1}}(S)) = [\![S]\!]([\![G]\!](W)\mathsf{args}^{\mathsf{G}}(S)) \ ,$$

  and $\mathsf{correct}(S)$ holds.

All the subcircuits of A_n have been included into A_{n+1}, either in their initial form, or united together with some other circuits. We have $\bigcup_{S \in A_{n+1}} = A_n$, and hence $I(A_n) = I(A_{n+1})$. Since for all $S \in A_n$ we have $\mathsf{correct}(S) = 1$, it should be $\forall W : I(A_n) \to \mathit{Val}, [\![A_n]\!](W) = [\![A_{n+1}]\!](W)$. By transitivity, $I(\mathsf{G}) = I(A_{n+1})$, and $\forall W : I(\mathsf{G}) \to \mathit{Val}, [\![G]\!](W) = [\![A_{n+1}]\!](W)$. □

6.5.5 Correctness of the Greedy Algorithms

We need to show that the algorithms of Section 6.4.5 are terminating, i.e. if we already have fixed a clique greedily, it will not prevent the other gates from being taken at all. Intuitively, whatever cliques we have fixed, as far as they do not contradict each other, all the other gates may be at least added as singleton cliques without causing any problems. We state it in the following lemma.

Lemma 6.4. *The function* Greed *terminates for any set of gates G with properly defined predicates* $\mathsf{fusable}^{\mathsf{G}}$, $\mathsf{pred}^{\mathsf{G}}$, *and* $\mathsf{cpred}^{\mathsf{G}}$, *producing a partitioning Cs of gates G that satisfy* $\mathsf{goodClique}(C, Cs) = \mathsf{true}$ *for any $C \in Cs$.*

Proof. The loops of Algorithm 5 and Algorithm 6 that wait until the set of gates Gs does not decrease anymore (both on lines 2-4) will definitely terminate since the size of a finite set cannot decrease infinitely. The loop of Algorithm 4 on lines 2-4) terminates unless $C = \emptyset$, which is not the case since the set Cs' constructed by largestClique does not contain empty sets, since at least $i \in G$ is contained in each clique.

Another source of possible non-terminations are the searches for a solution that satisfies goodClique in the functions largestClique, matching and maxMatching. In particular, if a suitable clique is not found, then these algorithms may return undefined values instead of valid sets of cliques. Since matching may always take $G_1 = G_2$, and the partitioning to singleton gates is also included into Css of maxMatching and Cs' of largestClique, it suffices to show that at least singleton gates are accepted as valid cliques. This should be always possible, regardless of the cliques that have already been fixed before.

Let Cs be the set of cliques collected so far. Each strategy fixes a clique only if it has passed the goodClique test at some point. Assume that $\mathsf{goodClique}(C, Cs) = \mathsf{true}$ for all $C \in Cs$. Now we want to add a clique $\{g\}$ to Cs, where $g \in G$ is an arbitrary gate. We need to show that $\mathsf{goodClique}(C, Cs \cup \{g\}) = \mathsf{true}$ for all $C \in Cs \cup \{g\}$.

1. First, we show that $\mathsf{goodClique}(\{g\}, Cs \cup \{g\}) = \mathsf{true}$ holds. Suppose by contrary that it is impossible. This may happen in the following cases:

   (a) For some $g \in \{g\}$, $\mathsf{fusable}^{\mathsf{G}}(g, g) = \mathsf{false}$. By definition of fusable, we always have $\mathsf{fusable}^{\mathsf{G}}(g, g) = \mathsf{true}$.

(b) It happens that $n_1 \geq n_2$ for the values $n_1 \leftarrow \mathsf{max}(\{0\} \cup PredLevels)$, $n_2 \leftarrow \mathsf{min}(\{N\} \cup SuccLevels)$, where we have $PredLevels \leftarrow \{\mathsf{level}(k) \mid g \in C, \mathsf{pred}^{\mathsf{G}}(g,k) \vee \mathsf{cpred}^{\mathsf{G}}(g,k) \wedge |C| > 1\}$, and also $SuccLevels \leftarrow \{\mathsf{level}(k) \mid g \in C, \mathsf{pred}^{\mathsf{G}}(k,g) \vee \mathsf{cpred}^{\mathsf{G}}(k,g) \wedge k \in C', C' \in Cs, |C'| > 1\}$. This means that there are some gates k, j belonging to cliques C_k and C_j such that, $\mathsf{level}(j) \leq \mathsf{level}(k)$, $\mathsf{pred}^{\mathsf{G}}(g,k) = 1$, and either $\mathsf{pred}^{\mathsf{G}}(j,g) = 1$ or $\mathsf{cpred}^{\mathsf{G}}(j,g) = 1, |C_j| > 1$. For $\mathsf{cpred}^{\mathsf{G}}(g,k) = 1$ case, $|C| > 1$ never holds since $C = \{g\}$. However, by transitivity of $\mathsf{pred}^{\mathsf{G}}$ and $\mathsf{cpred}^{\mathsf{G}}$, the statements $\mathsf{pred}^{\mathsf{G}}(j,k) = 1$ and $\mathsf{cpred}^{\mathsf{G}}(j,k) = 1, |C_j| > 1$) are also true, which contradicts the fact that $\mathsf{goodClique}(C_j, Cs) = \mathsf{true}$ and $\mathsf{goodClique}(C_k, Cs) = \mathsf{true}$.

Since $\mathsf{goodClique}(\{g\}, Cs \cup \{g\}) = \mathsf{true}$, we assign $\mathsf{level}(g) = n_g$ for some n_g as a side-effect.

2. After having assigned $\mathsf{level}(g) = n_g$, we need to prove that it has not broken the correctness of any old cliques, i.e. $\mathsf{goodClique}(C, Cs \cup \{g\}) = \mathsf{true}$ holds for all $C \in Cs$, where Cs is the set of old cliques. Since $\mathsf{goodClique}(C, Cs) = \mathsf{true}$ holds due to induction hypothesis (adding a gate does not modify any predicates concerning the cliques that are already fixed), it remains to prove that we have $\mathsf{goodClique}(C, \{g\}) = \mathsf{true}$.

   Let $C \in Cs$. Let n_1 and n_2 be the sizes of old sets $Predlevels$ and $SuccLevels$ before adding $\{g\}$. After adding $\{g\}$, there may be now more values that may get into these sets. Without loss of generality, let g be some successor of C. The minimal successor level is now $n_2' = \min(n_g, n_2)$. Since we have already shown that $\mathsf{goodClique}(\{g\}, Cs \cup \{g\}) = \mathsf{true}$ holds, we have assigned $\mathsf{level}(g) = n_g > \mathsf{level}(k)$ for all $k \in Cs$, so $n_g > (n_1 + n_2)/2$, and we have $\mathsf{level}(k) = n_1 < (n_1 + \min(n_g, n_2))/2 < \min(n_g, n_2) = n_2'$, so $n_1' < n_2$, and $\mathsf{goodClique}(C, Cs \cup \{g\}) = \mathsf{true}$. □

6.5.6 Correctness of the Reduction to ILP

We prove the correctness of the building block constraints of Section 6.4.6. We show which variables are implicitly binary. Finally, we prove the feasibility of ILP task.

Lemma 6.5. *If* $x \in \{0,1\}$, $y \leq C \in \mathbb{R}$, *then*
$\mathcal{P}(C,x,y,z) = \mathsf{true} \iff z = x \cdot y$.

Proof. The correctness and completeness of these constraints can be easily verified by case distinction on x for any $y \leq C$.

1. Substitute $x = 0$ into the constraints:

   - $y - z \leq C$,
   - $-y + z \leq C$,
   - $-z \geq 0$.

   The last constraint uniquely defines $z = 0$. The first two constraints are true since $y \leq C$.

2. Substitute $x = 1$ into the constraints:

   - $y - z \leq 0$,
   - $-y + z \leq 0$,
   - $C - z \geq 0$.

   The first two constraints uniquely define $z = y$. The last constraint is true since $z = y \leq C$. □

Lemma 6.6. *If $x \in \{0, 1\}$ for all $x \in \mathcal{X}$, $0 \leq y \leq C \in \mathbb{R}$, then $\mathcal{F}(A, C, \mathcal{X}, y) = \mathsf{true}$ iff $y = 1 \iff \sum_{x \in \mathcal{X}} x \geq A$.*

Proof. $\implies$ Let the constraints be satisfied. By Lemma 6.5, we have $z_x = x \cdot y$ for all $x \in \mathcal{X}$. Substituting z_x into the last two constraints, we get:

- $A \cdot y - y \cdot \sum_{x \in \mathcal{X}} x \leq 0$,
- $\sum_{x \in \mathcal{X}} x - y \cdot \sum_{x \in \mathcal{X}} x + (A - 1)y \leq (A - 1)$.

We can rewrite these constraints as

- $y(A - \sum_{x \in \mathcal{X}} x) \leq 0$,
- $\sum_{x \in \mathcal{X}} x(1 - y) \leq (A - 1)(1 - y)$.

We get that $\sum_{x \in \mathcal{X}} x \geq A$ unless $y = 0$, and $\sum_{x \in \mathcal{X}} x \leq (A - 1)$ unless $y = 1$. Hence the constraints are satisfiable only if $y \in \{0, 1\}$. If $y = 1$, then $\sum_{x \in \mathcal{X}} x \geq A$, and if $y = 0$, then $\sum_{x \in \mathcal{X}} x \leq (A - 1)$.

$\impliedby$ Let $y = 1 \iff \sum_{x \in \mathcal{X}} x \geq A$. In order to satisfy the constraints $\mathcal{P}(C, y, x, z_x)$, by Lemma 6.5 we take $z_x = x \cdot y$. We show by case distinction that the remaining two constraints are satisfied for both $y = 0$ and $y = 1$.

1. Let $y = 0$.

   - $A \cdot 0 - 0 \cdot \sum_{x \in \mathcal{X}} x \leq 0$,
   - $\sum_{x \in \mathcal{X}} x - 0 \cdot \sum_{x \in \mathcal{X}} x + (A - 1) \cdot 0 \leq (A - 1)$.

The first constraint is always true. The second one is satisfied if $\sum_{x\in\mathcal{X}} x \leq (A-1)$, which is equivalent to $\sum_{x\in\mathcal{X}} x < A$ since $x \in \{0,1\}$.

2. Let $y = 1$.

   - $A - \sum_{x\in\mathcal{X}} x \leq 0$,
   - $\sum_{x\in\mathcal{X}} x - \sum_{x\in\mathcal{X}} x + (A-1) \leq (A-1)$.

   The second constraint is true. The first one is satisfied if $\sum_{x\in\mathcal{X}} x \geq A$. □

Lemma 6.7. *If $z \in \{0,1\}$, $0 \leq x, y \leq C \in \mathbb{R}$, then* $\mathcal{L}(C, A, y, x, z) = \mathsf{true}$ *iff* $z = 1 \implies (x - y) \geq A$.

Proof. The correctness and completeness of the constraint $(C+A)\cdot z+(y-x) \leq C$ can be easily verified by case distinction on z for any $0 \leq x, y \leq C$.

1. Substitute $z = 0$: get $y - x \leq C$, which is always true for any $0 \leq x, y \leq C$.

2. Substitute $z = 1$: get $(C + A) + y - x \leq C$, which is equivalent to $A + y - x \leq 0$, or $x - y \geq A$. □

Lemma 6.8. *For $(A, \vec{b}, \vec{c}, \mathcal{I}) = T^{\rightarrow}_{ILP}(G, X, Y)$, the following variables of any feasible solution of $(A, \vec{b}, \vec{c}, \mathcal{I})$ are binary:*

g_i^j	*for $j, i \in G$;*
b_j c_j, u_j,	*for $j \in G$;*
fx_ℓ^{jk}	*for $j \in G$, $k \in V(\mathsf{G})$, $\ell \in [\mathsf{arity}^{\mathsf{G}}(j)]$;*
$e_{i\ell}^{jk}$	*for $j, i \in G$, $k \in V(\mathsf{G})$, $\ell \in [\mathsf{arity}^{\mathsf{G}}(j)]$;*
fg^{ji}, fg_ℓ^{ji}	*for $j, i \in G$, $\ell \in [\mathsf{arity}^{\mathsf{G}}(j)]$;*
s_ℓ^j	*for $j \in G$ $\ell \in [\mathsf{arity}^{\mathsf{G}}(j)]$.*

Proof. By Lemma 6.5, if $x \in \{0,1\}$ and $y \leq C$, then $\mathcal{P}(C, x, y, z)$ ensures $z \in \{0,1\}$. By Lemma 6.6, if $\forall x \in \mathcal{X} : \ x \in \{0,1\}$, then $\mathcal{F}(A, \mathcal{X}, y)$ ensures $y \in \{0,1\}$. We use these properties to propagate binariness.

We will make the proof for all types of variables one by one.

- The condition $g_i^j \in \{0,1\}$ is explicit in the ILP description (the set $\mathcal{I}$).
- By constraints (6f), $c_i \in \{0,1\}$ since $g_i^j \in \{0,1\}$, $d_j = (1 - g_j^j) \in \{0,1\}$.
- By constraints (7a), $fx_\ell^{jk} \in \{0,1\}$ since $g_i^j \in \{0,1\}$.
- By constraints (7b), $e_{i\ell}^{jk} \in \{0,1\}$ since $g_i^j \in \{0,1\}$ and $fx_\ell^{jk} \in \{0,1\}$.
- By constraints (7c), for $k \in I(\mathsf{G})$, $fg_\ell^{ji} \in \{0,1\}$ since $e_{i\ell}^{jk} \in \{0,1\}$.

- By constraints (7d), for $k \notin I(\mathsf{G})$, $fg_\ell^{ji} = fx_\ell^{jk}$ and hence $\in \{0,1\}$
- By constraints (7e), $s_\ell^j \in \{0,1\}$ since $fg_\ell^{jk} \in \{0,1\}$.
- By constraint (8a), $fg^{jk} \in \{0,1\}$ since $fg_\ell^{jk} \in \{0,1\}$.
- By constraints (8b), $u^j \in \{0,1\}$ since $fg^{jk} \in \{0,1\}$.
- By constraints (9a), $t^j \in \{0,1\}$ since $s_\ell^j \in \{0,1\}$. Hence, by constraints (9b-9c), $t_i^j \in \{0,1\}$. Note that, for a fixed i, exactly one $g_i^j = 1$ due to constraints (2). By definition of $\mathcal{P}$, we have $t_i^j = g_i^j \cdot t^j$, and hence at most one $t_i^j = 1$. By constraints (9d), $b_i = \sum_{j \in G} t_i^j$, and so $b_i \in \{0,1\}$. □

Proof of feasibility of the integer programming task (Theorem 6.1) Let $(A, \vec{b}, \vec{c}, \mathcal{I})$ be the mixed integer linear programming task. It has a solution iff the system $A\vec{x} \leq \vec{b}$ has at least one solution, assuming that $\forall i \in \mathcal{I} : \; x_i \in \{0,1\}$. We show that any solution in which $g_j^j = 1$ for all $j \in G$ and $g_i^j = 0$ for all $j, i \in G$, $i \neq j$, is feasible. Intuitively, this means that it is always possible not to fuse any gates, leaving the circuit as it is.

By Lemma 6.5 and Lemma 6.6 the constraints $\mathcal{P}(C, x, y, z)$ and $\mathcal{F}(A, \mathcal{X}, z)$ are always satisfied if z is a new variable that has not been present in any other constraints before at this point.

Let $\forall j \in G : \; g_j^j = 1$, and $\forall j, i \in G, i \neq j : \; g_i^j = 0$. We show one by one, that all the constraints are satisfied.

1. $g_i^j + g_k^j \leq 1$ for $i, k \in G$, $\neg\mathsf{fusable}^{\mathsf{G}}(i,k)$.
   Since $\mathsf{fusable}^{\mathsf{G}}(i,i)$ holds for all i, here we have $i \neq k$, and never get the case $g_j^j + g_j^j \leq 1$. For all the other $g_i^j + g_k^j$ at least one term is 0.

2. $\sum_{j=1}^{|G|} g_i^j = 1$ for all $i \in G$.
   For any $i \in G$, the only j such that $g_i^j = 1$ is $j = i$.

3. $g_i^j = 0$ if $\mathsf{op}^{\mathsf{G}}(i) \neq \mathsf{op}^{\mathsf{G}}(j)$.
   We have $g_i^j = 1$ only if $i = j$, but then $\mathsf{op}^{\mathsf{G}}(i) = \mathsf{op}^{\mathsf{G}}(j)$.

4. $g_j^j - g_i^j \geq 0$ for all $i \in G$, $j \in G$.
   This is true since $g_j^j = 1$ and all g_i^j are binary by Lemma 6.8.

5. $g_j^j = 1$ for all j such that $cost(\mathsf{op}^{\mathsf{G}}(j)) = 0$.
   All $g_j^j = 1$ anyway.

6. We show that a possible evaluation of ℓ_i is the topological ordering of gates in the initial circuit.

(a) $\ell_i - \ell_k \geq 1$ for all $i, k \in G, \mathsf{pred}^{\mathsf{G}}(i,k)$;
the constraint is satisfied by definition of $\mathsf{pred}^{\mathsf{G}}(i,k)$ and the fact that we use topological ordering which assigns a strictly smaller level to the gate predecessors.

(b) The constraints $\mathcal{L}(|G|, 0, \ell_i, \ell_j, g_i^j)$ and $\mathcal{L}(|G|, 0, \ell_j, \ell_i, g_i^j)$ are satisfied since we only have $g_j^j = 1$, and $\ell_j = \ell_j$ is trivially satisfied.

(c) $\ell_i \geq 0, \ell_i \leq |G|$.
This holds by definition of topological order: it assigns a unique number to each gate.

(d) $d_j = (1 - g_j^j)$;
Satisfied since d_j is a newly introduced variable.

(e) $\mathcal{F}(1, \{d_j\} \cup \{g_i^j \mid i \in G, i \neq j\}, c_j)$;
Satisfied since c_j is a newly introduced variable. Namely, since $g_j^j = 1$, and $g_i^j = 0$ for $i \neq j$, we have $c_j = 0$ for all j.

(f) $\mathcal{L}(|G|, 1, \ell_i, \ell_k, c_i)$ for $\mathsf{cpred}^{\mathsf{G}}(i,k)$.
By Lemma 6.7, $c_i = 1$ implies $\ell_i - \ell_j \geq 0$. Since we have $c_i = 0$, the implication is trivially true.

7. The constraints (7) are satisfied due to introducing the new variables fx_ℓ^{jk}, $e_{i\ell}^{jk}$, fg_ℓ^{ji}, s_ℓ^j, and sc_ℓ^j.

8. The constraints (8) are satisfied due to the new variables fg^{jk}, u^j, uc^j.

9. The constraints (9) are satisfied due to the new variables t^j, t_i^j, b_i. □

6.5.7 Correctness of the Circuit Transformation

We prove the correctness of $T_{ILP}^{\leftarrow}$ (defined in Section 6.4.7) transforming the ILP solution to a circuit. We estimate the cost of the resulting circuit, and show that $T_{ILP}^{\leftarrow}$ can be as well applied to a circuit after using any of the greedy algorithms of Section 6.5.5 instead of an ILP task.

Proof of Correctness of $T_{ILP}^{\leftarrow}$ (Theorem 6.2)

Let $\mathsf{G} = (G, X, Y)$ be the initial circuit. Let $\mathsf{G}' = (G', X', Y')$ be the transformed circuit. We show that $[\![\mathtt{eval}(G, X, Y)]\!]\, s = [\![\mathtt{eval}(G', X', Y')]\!]\, s$. In order to make the proof easier, let us rewrite the expressions according to their definitions:

- $[\![\mathtt{eval}(G, X, Y)]\!]\, s = \mathtt{upd}(Y \circ [\![G]\!](s \circ X), s)$;
- $[\![\mathtt{eval}(G', X', Y')]\!]\, s = \mathtt{upd}(Y' \circ [\![G]\!](s \circ X'), s)$.

Since Algorithm 7 defines $Y' \leftarrow Y$ on line 2, and only the range of Y' is modified on line 16, we have $\mathsf{Dom}(Y) = \mathsf{Dom}(Y') =: \mathcal{Y}$. It suffices to prove that

$$\forall y \in \mathcal{Y} : \ [\![G]\!](s \circ X)(Y(y)) = [\![G']\!](s \circ X')(Y'(y)) \ .$$

For all j, we have defined $Y' = Y[i \leftarrow j \mid g_i^j = 1]$ on line 16. Since there is exactly one j such that $g_i^j = 1$ (by constraints (2) of Section 6.4.6), we should actually prove

$$\forall i, j \in G, i, j \in \mathcal{Y} : \ (g_i^j = 1) \implies [\![G]\!](s \circ X)(i) = [\![G']\!](s \circ X')(j) \ . \quad (6.1)$$

Since all the output wires of (G, X, Y) are evaluated in any case (by circuit definition), for each i such that $i \in \mathcal{Y}$ we may add an additional assumption $[\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1$, where ϕ_i^{G} is the weakest precondition of evaluating i. This will be useful during the proof by induction. We may now replace the assumption $i, j \in \mathcal{Y}$ with $[\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1$ in (6.1), proving a less general result

$$\forall i, j \in G : \ ([\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \wedge g_i^j = 1) \implies [\![G]\!](s \circ X)(i) = [\![G']\!](s \circ X')(j) \ . \quad (6.2)$$

We prove this statement by induction on the number of the first j topologically ordered cliques that have already been processed. More precisely, for the clique ordering we use the variables ℓ_i from the constraints (6).

First, we prove that the gate ordering defined by ℓ_i creates no cycles. Namely, we prove that if a gate g_j is computed on the level ℓ_j, then each its argument has been computed on the level ℓ_k for $\ell_k < \ell_j$.

Lemma 6.9. *The following claims hold.*

- *For all $j \in G$, $k \in \mathsf{args}^{\mathsf{G}}(j)$: $\ell_k < \ell_j$.*
- *For all $j, i \in G$: if $g_i^j = 1$, then $\ell_i = \ell_j$.*
- *For all $i \in G$, $k \in \mathsf{args}^{\mathsf{G}}(\phi_i^{\mathsf{G}})$: if $g_i^j = 1$ for some $j \neq i$, then $\ell_k < \ell_i$.*

Proof. Recall the constraints 6:

(a) $\ell_i - \ell_k \geq 1$ for all $i, k \in G, \mathsf{pred}^{\mathsf{G}}(i, k)$;
since we define $\mathsf{pred}^{\mathsf{G}}(i, k)$ for all $k \in \mathsf{args}^{\mathsf{G}}(i)$, this constraint ensures that $\forall j \in G, k \in \mathsf{args}^{\mathsf{G}}(j) : \ell_k < \ell_j$.

(b-d) $\mathcal{L}(|G|, 0, \ell_i, \ell_j, g_i^j)$, $\mathcal{L}(|G|, 0, \ell_j, \ell_i, g_i^j)$, and $0 \leq \ell_i \leq |G|$ for all $i, j \in G$;
since g_i^j are binary variables, by Lemma 6.7 this ensures that, if $g_i^j = 1$, then $\ell_i \leq \ell_j$ and $\ell_j \leq \ell_i$, so $\ell_i = \ell_j$.

(e-f) $d_j = (1 - g_j^j)$ and $\mathcal{F}(1, \{d_j\} \cup \{g_i^j \mid i \in G, i \neq j\}, c_j)$ for all $j \in G$;
By Lemma 6.6, the constraints ensure that $c_j \in \{0, 1\}$, and that $c_j = 1$ iff either $g_i^j = 1$ for some $j \neq i$, or there is k s.t $g_j^k = 1$.

(g) $\mathcal{L}(|G|, 1, \ell_i, \ell_k, c_i)$ for all $i, k \in G$, $\mathsf{cpred}^{\mathsf{G}}(i, k)$;
By definition, $\mathsf{cpred}^{\mathsf{G}}(i, k) = 1$ iff $k \in \mathsf{args}^{\mathsf{G}}(\phi_i^{\mathsf{G}})$. By Lemma 6.7, since c_i is a binary variable, $c_i = 1$ implies $\ell_i - \ell_k \geq 1$, which is $\ell_k < \ell_i$, and $c_i = 1$ is implied by $g_i^j = 1$, $i \neq j$ according to the previous constraints. □

Let G_j be the subcircuit of G consisting just of the gates belonging to the first j cliques ordered by ℓ_k (i.e. $G_j = \{C_k \mid \ell_k \leq \ell_j\}$ where $C_k = \{i \mid g_i^k = 1\}$). Let G'_j be the subcircuit of G' obtained after processing the first j cliques by Algorithm 7. In this way, if there are m cliques in total, then $G' = G'_m$, and $G = G_m$.

Base: $G_0 = \emptyset$, and the statement (6.2) is trivially true.

Step: Suppose that we are adding the clique C_j to the subcircuit G_{j-1}. By the induction hypothesis, the statement (6.2) already holds for all $i, j \in G \setminus C_j$, so it suffices to prove that

$$\forall i \in C_j : (\llbracket \phi_i^{\mathsf{G}} \rrbracket \llbracket G \rrbracket (s \circ X) = 1 \wedge g_i^j = 1) \implies \llbracket G_j \rrbracket (s \circ X)(i) = \llbracket G'_j \rrbracket (s \circ X')(j) \ . \quad (6.3)$$

Since by definition of C_j we have $\forall i \in C_j : g_i^j = 1$, we may simplify (6.3) and prove

$$\forall i \in C_j : \llbracket \phi_i^{\mathsf{G}} \rrbracket \llbracket G \rrbracket (s \circ X) = 1 \implies \llbracket G_j \rrbracket (s \circ X)(i) = \llbracket G'_j \rrbracket (s \circ X')(j) \ . \quad (6.4)$$

Let $i \in C_j$. Let $t = \mathsf{arity}^{\mathsf{G}}(j)$. Before the transformation, for all i s.t $g_i^j = 1$, due to the constraint (2), which states that a gate belongs to exactly one clique, there should have been exactly one gate $(i, \mathsf{op}^{\mathsf{G}}(i), [k_1, \ldots, k_t])$ in C_j, and, due to constraints (3), which state that a gate has the same operation as the clique representative, $\mathsf{op}^{\mathsf{G}}(i) = \mathsf{op}^{\mathsf{G}}(j)$.

First, let $\mathsf{op}^{\mathsf{G}}(j) \neq oc$. In this case, new the input wires of j are exactly the new variables v_ℓ^j. By definition,

$$\llbracket G'_j \rrbracket (s \circ X')(j) = \llbracket \mathsf{op}^{G'}(j) \rrbracket (\llbracket G'_j \rrbracket (s \circ X')(v_1^j), \ldots, \llbracket G'_j \rrbracket (s \circ X')(v_t^j)) \ ,$$

and

$$\llbracket G_j \rrbracket (s \circ X)(i) = \llbracket \mathsf{op}^{\mathsf{G}}(j) \rrbracket (\llbracket G_j \rrbracket (s \circ X)(k_1), \ldots, \llbracket G_j \rrbracket (s \circ X)(k_t)) \ .$$

Since $\mathsf{op}^{\mathsf{G}}(j) \neq oc$, we have $\mathsf{op}^{\mathsf{G}}(j) = \mathsf{op}^{G'}(j)$, and it suffices to show that the arguments of $[\![\mathsf{op}^{\mathsf{G}}(j)]\!]$ in both cases are the same, i.e

$$\forall \ell \in [t] : \ ([\![\phi_i^{\mathsf{G}}]\!]\, s = 1) \implies [\![G_j]\!](s \circ X)(k_\ell) = [\![G'_j]\!](s \circ X')(v_\ell^j) \ . \quad (6.5)$$

Let $\mathcal{K}_\ell^j$ denote the set $\mathcal{K}$ formed by Algorithm 8 when called by Algorithm 7 with the input $\mathcal{B}_\ell^j$. We will prove statement (6.5) for different cases of $|\mathcal{K}_\ell^j|$.

1. If $|\mathcal{K}_\ell^j| \leq 1$, then Algorithm 8 creates a gate (v_ℓ^j, id, w_ℓ^j) for $\{w_\ell^j\} \leftarrow \mathcal{K}_\ell^j$. Since a non-empty clique C_j has at least one ℓ-th input (the one belonging to the gate j), we have $|\mathcal{K}_\ell^j| = 1$, and so such w_ℓ^j exists, and the definition of v_ℓ^j is correct. Since w_ℓ^j is the only ℓ-th input of C_j, and k_ℓ is the ℓ-th input of some gate of C_j by definition, we have $w_\ell^j = k_\ell$, and since id does not modify the value, we have $[\![G'_j]\!](s \circ X')(v_\ell^j) = [\![G'_j]\!](s \circ X')(w_\ell^j)$.

   (a) If $w_\ell^j \in I(\mathsf{G})$, then trivially $w_\ell^j \in G'_i$ since Algorithm 7 keeps all the input gates of the initial circuit on line 2.

   (b) If $w_\ell^j \notin I(\mathsf{G})$, then there exists $u \in C_i$ such that $u = \mathsf{args}^{\mathsf{G}}(j)[\ell]$. We have $\mathsf{pred}^{\mathsf{G}}(j, u)$, and by Lemma 6.9, $\ell_u < \ell_j$. Hence $w_\ell^j \in G'_u \subseteq G'_j$.

   We get that the values w_ℓ^j chosen by Algorithm 7 satisfy $w_\ell^j \in G'_j$. Since $w_\ell^j = k_\ell$, we have $[\![G_j]\!](s \circ X)(k_\ell) = [\![G'_j]\!](s \circ X')(w_\ell^j) = [\![G'_j]\!](s \circ X')(v_\ell^j)$.

2. If $|\mathcal{K}_\ell^j| > 1$, Algorithm 8 defines a subcircuit $(G_\ell^{jk}, X_\ell^{jk}, Y_\ell^{jk})$ computing $b_\ell^{jk} := \sum_{i \in \mathcal{I}_\ell^{jk}} \phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'} = 1]$, where, by definition of $\mathcal{B}_\ell^j$,

$$\begin{aligned} \mathcal{I}_\ell^{jk} \ = \ & \{i \mid k \in I(\mathsf{G}), k = \mathsf{args}^{\mathsf{G}}(i)[\ell]\} \\ & \cup \{i \mid k \notin I(\mathsf{G}), k \in C_u, u = \mathsf{args}^{\mathsf{G}}(i)[\ell]\} \ . \end{aligned}$$

   The gate names of $(G_\ell^{jk}, X_\ell^{jk}, Y_\ell^{jk})$ are fresh, so there are no name conflicts with the gate names of G'_{j-1}. Since all the gates are new and do not belong to G, we do not need to prove statement (6.4) for them.

   The following lemma proves the correctness of computing the conditions of the newly introduced *oc* gates. By definition, we collect all the gates that use the k as the ℓ-th argument into $\mathcal{I}_\ell^{jk}$, and hence the condition for choosing $i \in \mathcal{I}_\ell^{jk}$ should be $\sum_{i \in \mathcal{I}_\ell^{jk}} \phi_i^{\mathsf{G}}$.

Lemma 6.10. *Let* $|\mathcal{K}_\ell^j| > 1$. *If* $[\![\phi_k^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1$, *then*

$$[\![G'_j]\!](s \circ X')b_\ell^{jk} = [\![\sum_{i \in \mathcal{I}_\ell^{jk}} \phi_i^{\mathsf{G}}]\!]\,[\![G_j]\!](s \circ X) \ .$$

Proof. Since $b_\ell^{jk} = \sum_{i \in \mathcal{I}_\ell^{jk}} \phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'} = 1]$, it suffices to show that

$$[\![\phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'} = 1]]\!]\,[\![G'_j]\!](s \circ X') = [\![\phi_i^{\mathsf{G}}]\!]\,[\![G_j]\!](s \circ X) \ .$$

We can rewrite it as

$$[\![\phi_i^{\mathsf{G}}]\!]\,[\![G'_j]\!](s \circ X')[i' \leftarrow j' \mid g_{i'}^{j'} = 1] = [\![\phi_i^{\mathsf{G}}]\!]\,[\![G_j]\!](s \circ X) \ .$$

For all variables i' of ϕ_i^{G}, we have $\mathsf{cpred}^{\mathsf{G}}(i, i')$. By Lemma 6.9, $\ell_{i'} < \ell_i$, and by the induction hypothesis, $[\![\phi_{i'}^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \implies [\![G_j]\!](s \circ X)(i') = [\![G'_j]\!](s \circ X')(j')$.

In order to apply the hypothesis, we need $[\![\phi_{i'}^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1$ to hold, but we have $[\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1$. Since each i' is involved in the computation of i and is its predecessor, we have

$$[\![\phi_{\phi_i^{\mathsf{G}}}^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \implies [\![\phi_{i'}^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \ .$$

By assumption, $[\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1$. Assuming that ϕ_i^{G} has to be evaluated in order to get $[\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X)$, this implies that $[\![\phi_{\phi_i^{\mathsf{G}}}^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1$ also holds. We get an implication

$$[\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \implies [\![\phi_{i'}^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \ .$$

We get $[\![G_j]\!](s \circ X)(i') = [\![G'_j]\!](s \circ X')(j')$ for all i' that define the value of ϕ_i^{G}. Hence

$$[\![\phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'} = 1]]\!]\,[\![G'_j]\!](s \circ X') = [\![\phi_i^{\mathsf{G}}]\!]\,[\![G_j]\!](s \circ X) \ . \qquad \square$$

Algorithm 7 defines a new gate $(v_\ell^j, oc, [b_\ell^{jk}, k]_{k \in \mathcal{K}_\ell^j})$ for a new variable v_ℓ^j that has not been used anywhere else before. We have

$$[\![G_j]\!](s \circ X)(v_\ell^j) = [\![oc]\!]([b_\ell^{jk}, k]_{k \in \mathcal{K}_\ell^j}) \ .$$

By constraints (1) and the definition of $\mathsf{fusable}^{\mathsf{G}}$, the weakest preconditions of the gates inside one clique are mutually exclusive, and hence for any s, at most one of $[\![G'_j]\!](s \circ X')(b_\ell^{jk})$ is 1, so this is a valid instance of oc.

We need to prove the equality $[\![G'_j]\!](s \circ X')(v^j_\ell) = [\![G_j]\!](s \circ X)(k_\ell)$, where k_ℓ is the ℓ-th input of i, on the assumption $[\![\phi^{\mathsf{G}}_i]\!]\,[\![G_j]\!](s \circ X) = 1$. Since $k_\ell \in \mathcal{K}^j_\ell$, there is a pair $(b^{jk_\ell}_\ell, k_\ell)$ in the oblivious choice.

By Lemma 6.10 we have

$$[\![G'_j]\!](s \circ X')(b^{jk_\ell}_\ell) = [\![\sum_{i' \in \mathcal{I}^{jk_\ell}_\ell} \phi^{\mathsf{G}}_{i'}]\!]\,[\![G_j]\!](s \circ X) \ .$$

By definition of $\mathcal{I}^{jk_\ell}_\ell$, k_ℓ is a predecessor of all $i' \in \mathcal{I}^{jk_\ell}_\ell$, so we have $([\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![\phi^{\mathsf{G}}_{k_\ell}]\!]\,[\![G_j]\!](s \circ X) = 1)$ for all $i' \in \mathcal{I}^{jk_\ell}_\ell$ and since $([\![\phi^{\mathsf{G}}_{k_\ell}]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![b^{jk_\ell}_\ell]\!]\,s = 1)$, we also have $([\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![b^{jk_\ell}_\ell]\!]\,s = 1)$, in particular $([\![\phi^{\mathsf{G}}_i]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![b^{jk_\ell}_\ell]\!]\,s = 1)$.

Hence it suffices to prove $\forall \ell : \ ([\![b^{ji_\ell}_\ell]\!]\,s = 1) \implies [\![G'_j]\!](s \circ X')(v^j_\ell) = [\![G_j]\!](s \circ X)(k_\ell)$. If $[\![b^{jk_\ell}_\ell]\!]\,s = 1$, then $[\![G'_j]\!](s \circ X')(v^j_\ell) = [\![G'_j]\!](s \circ X')(k_\ell)$ by definition of oc. Since $k_\ell \in C_{i'}$ for some $i' < j$, we have $[\![G'_j]\!](s \circ X')(k_\ell) = [\![G'_{j-1}]\!](s \circ X')(k_\ell)$, and due to $([\![\phi^{\mathsf{G}}_i]\!]\,s = 1) \implies ([\![\phi^{\mathsf{G}}_{k_\ell}]\!]\,s = 1)$, by the induction hypothesis equal to $[\![G_{j-1}]\!](s \circ X)(k_\ell) = [\![G_j]\!](s \circ X)(k_\ell)$.

If $\mathsf{op}^{\mathsf{G}}(j) \neq oc$, then a gate $(j, \mathsf{op}^{\mathsf{G}}(j), [v^j_1, \ldots, v^j_t])$ is added to the clique C_j. By the constraints (4) that define the clique representative, if the clique C_j is non-empty (there exists at least one $g^j_i = 1$), then definitely $j \in C_j$ ($g^j_j = 1$), and $\forall i \in G : \ g^i_j = 0$ due to constraints (2). Hence it is the only gate with the name j, so there are no name conflicts.

If $\mathsf{op}^{\mathsf{G}}(j) = oc$, then it may happen that $\mathsf{op}^{G'}(j)$ changes, and moreover, the inputs v^j_ℓ may be rearranged by the call to Algorithm 8. We need to prove $[\![G_j]\!](s \circ X)(i) = [\![G'_j]\!](s \circ X')(j)$ straightforwardly. The proof is analogous to the proof that $[\![G_j]\!](s \circ X)(k_\ell) = [\![G'_j]\!](s \circ X')(v^j_\ell)$, as we call the same Algorithm 8 here, and even a bit simpler since the conditions for the choices are the inputs of j, and not the weakest preconditions. □

Proof of the Cost of Transformed Circuit (Theorem 6.4)

First, we will prove some relations between different variables that are defined by the constraints. Let $\mathcal{K}^j_\ell$ and $\mathcal{K}^j$ denote the sets $\mathcal{K}$ formed by Algorithm 8 when called by Algorithm 7 with inputs $\mathcal{B}^j_\ell$ and $\mathcal{B}^j$ respectively.

Lemma 6.11. *For all $j \in G$, $\ell \in \mathsf{arity}^{\mathsf{G}}(j)$, $k \in V(\mathsf{G})$, we have:*

- $\mathcal{K}_\ell^j = \{k \mid fg_\ell^{jk} = 1\}$;
- $\mathcal{K}^j = \{k \mid fg^{jk} = 1\}$.

Proof. The proof is based on the fact that all the variables are binary (proven in Lemma 6.8), and on the definition of constraints $\mathcal{P}$ and $\mathcal{F}$ for binary inputs (proven in Lemma 6.5 and Lemma 6.6).

1. By definition of $\mathcal{F}$, since $g_i^j \in \{0,1\}$, $fx_\ell^{jk} \in \{0,1\}$, and $fx_\ell^{jk} = 1$ iff at least one $g_i^j = 1$ s.t $k = \mathsf{args}^{\mathsf{G}}(i)[\ell]$. Since $g_i^j = 1$ denotes $i \in C_j$, we get $fx_\ell^{jk} = 1$ iff $\exists i \in C_j : \; k = \mathsf{args}^{\mathsf{G}}(i)[\ell]$.

2. By definition of $\mathcal{P}$, since fx_ℓ^{jk} and g_k^i are binary, $e_{i\ell}^{jk} \in \{0,1\}$, and $e_{i\ell}^{jk} = 1$ iff $fx_\ell^{jk} \cdot g_k^i$. Since $g_k^i = 1$ denotes $k \in C_i$, we get $e_{i\ell}^{jk} = 1$ iff $k \in C_i$ and $k \in \mathsf{args}^{\mathsf{G}}(C_j)[\ell]$.

3. By definition of $\mathcal{F}$, for $i \in G$, since $e_{i\ell}^{jk}$ are binary, $fg_\ell^{jk} \in \{0,1\}$, and $fg_\ell^{ji} = 1$ iff there exists $k \in I(\mathsf{G})$ s.t $e_{i\ell}^{jk} = 1$, so there is some k that causes $k \in C_i$ and $k \in \mathsf{args}^{\mathsf{G}}(j)[\ell]$. We get $fg_\ell^{jk} = 1$ iff $\exists i \in C_j, u \in C_k : u = \mathsf{args}^{\mathsf{G}}(i)[\ell]$.

4. For $k \in I(\mathsf{G})$, define $fg_\ell^{jk} = fx_\ell^{jk}$. We get $fg_\ell^{jk} = 1$ iff $\exists i \in C_j : k = \mathsf{args}^{\mathsf{G}}(i)[\ell]$ for $k \in I(\mathsf{G})$, and $fg_\ell^{jk} = 1$ iff $\exists i \in C_j, u \in C_k : u = \mathsf{args}^{\mathsf{G}}(i)[\ell]$ for $k \notin I(\mathsf{G})$. By definition of $\mathcal{B}_\ell^j$ and $\mathcal{K}$, we have $fg_\ell^{jk} = 1$ iff $k \in \mathcal{K}_\ell^j$, so $\mathcal{K}_\ell^j = \{k \mid fg_\ell^{jk} = 1\}$.

5. By definition of $\mathcal{F}$, since $fg_\ell^{jk} \in \{0,1\}$, also $fg^{jk} \in \{0,1\}$, and $fg^{jk} = 1$ iff at least one $fg_\ell^{jk} = 1$ for $\ell \in 2\mathbb{N}$. We get $fg^{jk} = 1$ iff $\exists i \in C_j : k \in \mathsf{args}^{\mathsf{G}}(i)[\ell]$ for some $\ell \in 2\mathbb{N}$. By definition of $\mathcal{B}^j$ and $\mathcal{K}$, we have $fg^{jk} = 1$ iff $k \in \mathcal{K}^j$, so $\mathcal{K}^j = \{k \mid fg^{jk} = 1\}$. □

The following lemma intuitively proves that s_ℓ^j denotes if there are at least 2 choices for the ℓ-th input of C_j, and that these choices are captured by the set $\mathcal{K}_\ell^j$. Similarly, u^j denotes if there are at least 2 choices left for C_j in the case $\mathsf{op}^{\mathsf{G}}(j) = oc$, and that these choices are captured by $\mathcal{K}^j$.

Lemma 6.12. *If $g_j^j = 1$, then:*

1. $sc_\ell^j = |\mathcal{K}_\ell^j| - 1$;
2. $s_\ell^j = 0$ *iff* $|\mathcal{K}_\ell^j| = 1$;
3. $uc_\ell^j = |\mathcal{K}^j| - 1$;
4. $u^j = 0$ *iff* $|\mathcal{K}^j| = 1$.

Proof. The proof is based on the fact that all the variables are binary (proven in Lemma 6.8), and on the definition of constraints $\mathcal{P}$ and $\mathcal{F}$ for binary inputs (proven in Lemma 6.5 and Lemma 6.6).

By constraints (7d), $sc_\ell^j = \sum_{k \in V(\mathsf{G})} fg_\ell^{jk} - g_j^j$, and $s_\ell^j = 1$ iff at least two of fg_ℓ^{jk} are 1. By Lemma 6.11 we have $\mathcal{K}_\ell^j = \{k \mid fg_\ell^{jk} = 1\}$. It immediately follows that $sc_\ell^j = |\mathcal{K}_\ell^j| - 1$. If $s_\ell^j = 1$, we get $1 \neq |\mathcal{K}_\ell^j| > 2$.

If $s_\ell^j = 0$, there is at most one k such that $fg_\ell^{jk} = 1$, and it remains to prove that there is at least one such k. Suppose by contrary that there is no such k, and $|\mathcal{K}_\ell^j| = \emptyset$. Then it should be $\mathsf{args}^{\mathsf{G}}(C_j)[\ell] = \emptyset$. By definition, $\mathsf{args}^{\mathsf{G}}(C_j)[\ell] = \{k \mid i \in C_j, g_i^j = 1, k = \mathsf{args}^{\mathsf{G}}(i)[\ell]\}$. By assumption, we have $g_j^j = 1$, and since $j \in C_j$ and we have taken $\ell \in [\mathsf{arity}^{\mathsf{G}}(j)]$, at least $k = \mathsf{args}^{\mathsf{G}}(j)[\ell]$ is a suitable candidate.

The proof is analogous for u^j. □

The following lemma shows the relations of s_ℓ^j and b_i.

Lemma 6.13. *For all $i \in G$, if there exist $j \in G$, $\ell \in \mathsf{arity}^{\mathsf{G}}(j)$ s.t $s_\ell^j = 1$ and $g_i^j = 1$ for $j \neq i$, then $b_i = 1$.*

Proof. The proof is based on the fact that all the variables are binary (proven in Lemma 6.8), and on the definition of constraints $\mathcal{P}$ and $\mathcal{F}$ for binary inputs (proven in Lemma 6.5 and Lemma 6.6).

- By constraints (9a), if $s_\ell^j = 1$, then $t^j = 1$.
- By constraints (9b), if $t^j = 1$ and $g_i^j = 1$, for $j \neq i$, then $t_i^j = 1$.
- By constraints (9d), if $t_i^j = 1$ for $j \in G$, then $b_i = 1$. □

Proof of Theorem 6.4 We prove the cost for different kinds of gates, one by one.

- **Old non-*oc* gates:** Algorithm 7 adds to G' the gates j such that $g_j^j = 1$. While defining $\mathsf{fusable}^{\mathsf{G}}$, we agreed not to fuse the the gates whose complexity changes if their public inputs become private. Hence their total cost is $\mathsf{C}_g = \sum_{j=1,\mathsf{op}^{\mathsf{G}}(j) \neq oc}^{|G|} cost(\mathsf{op}^{\mathsf{G}}(j)) \cdot g_j^j$.
- **Old *oc* gates:** An old oc gate is replaced with an id by Algorithm 7 if $|\mathcal{K}^j| = 1$.
  - Let for $u^j = 1$. By Lemma 6.12, $u^j = 1$ implies $|\mathcal{K}^j| > 1$, causing Algorithm 7 to leave the oc gate into G'.

– Let for $u^j = 0$. By Lemma 6.12, $u^j = 0$ implies $|\mathcal{K}^j| = 1$, and Algorithm 7 replaces the oc gate with an id gate.

By Lemma 6.12, we have $uc^j = |\mathcal{K}^j| - g_j^j$, which is the number of choices that the old oc gate makes in the transformed graph. We have defined the cost of an oc gate as $cost(oc_{base}) + cost(oc_{step}) \cdot n$, where n is the number of choices that the oc gate makes. Hence the total cost of the new oc gates is $\mathsf{C}_{oc} = \sum_{j,\ell=1,1}^{|G|,\mathsf{arity}^{\mathsf{G}}(j)} cost(oc_{base}) \cdot u_\ell^j + cost(oc_{step}) \cdot uc_\ell^j$.

- **New** oc **gates:** An oc gate is created by Algorithm 7 if $|\mathcal{K}_\ell^j| > 1$.

  – Let for $s_\ell^j = 1$. By Lemma 6.12, $s_\ell^j = 1$ implies $|\mathcal{K}_\ell^j| > 1$, causing Algorithm 7 to construct a new oc gate v_ℓ^j that is included into G'.

  – Let for $s_\ell^j = 0$. By Lemma 6.12, $s_\ell^j = 0$ implies $|\mathcal{K}_\ell^j| = 1$, and Algorithm 7 does not construct an oc gate.

  By Lemma 6.12, we have $sc_\ell^j = |\mathcal{K}_\ell^j| - 1$, which is the number of choices that the new oc gate makes. The total cost of the new oc gates is $\mathsf{C}_{oc} = \sum_{j,\ell=1,1}^{|G|,\mathsf{arity}^{\mathsf{G}}(j)} cost(oc_{base}) \cdot s_\ell^j + cost(oc_{step}) \cdot sc_\ell^j$.

- **New** G_k **gates:** These gates are computing the conditions vb_k. For the old oc gates, vb_k are just some wire names, and the only other introduced operation is addition. Assuming that the addition operation is free (as it is in most practical SMC platforms), these gates may have some cost only if they are composed for a new oc gate. Let us assume that Algorithm 8 has been called by Algorithm 7 on some $\mathcal{B}_\ell^j$.

  For simplicity, we defined Algorithm 8 in such a way that it does not assign special default choices and uses only the weakest preconditions straightforwardly. However, since oc gates are correctly defined (by Theorem 6.2), all the vb_k arguments sum up either to 0 or 1. If they sum up to 0, then the weakest preconditions of all the choices are 0, and hence the output of oc does not matter anyway, so we may as well make one of the choices 1. If they do sum up to 1, then one of the vb_k arguments of the oc is actually a linear combination of the other its vb_k arguments. We may let Algorithm 7 to choose any k to be the default one. Without loss of generality, let it be $k = \mathsf{args}^{\mathsf{G}}(j)[\ell]$. For $g_j^j = 1$, we have $b_j = 0$ due to constraints (9a-9d). For $k \neq \mathsf{args}^{\mathsf{G}}(j)[\ell]$, Algorithm 7 does include the gates G_k into G'.

  We need to show that including G_k for $k \neq \mathsf{args}^{\mathsf{G}}(j)[\ell]$ into G' indeed implies including all ϕ_i^{G} for $i \neq j$. Suppose by contrary that there is some $i \neq j$ that only occurs in G_k for $k' = \mathsf{args}^{\mathsf{G}}(j)[\ell]$, regardless of choice of ℓ.

In other words, for all ℓ, the ℓ-th input of i is the ℓ-th input of j, and hence $\mathsf{args}^{\mathsf{G}}(i) = \mathsf{args}^{\mathsf{G}}(j)$. Since $\mathsf{op}^{\mathsf{G}}(i) = \mathsf{op}^{\mathsf{G}}(i)$, and the gates are unique, we have $i = j$. This contradicts the assumption $i \neq j$, so ϕ_i^{G} should have been used in at least one vb_k for $k \neq \mathsf{args}^{\mathsf{G}}(j)[\ell]$.

We get that the gates of ϕ_i^{G} are included into G' iff $b_j = 1$. For all $k \in C_j$, Algorithm 7 takes

$$vb_k = \sum_{(\phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'}=1],k)\in\mathcal{B}} \phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'} = 1] \ .$$

Substituting i' with j' does not affect the cost of ϕ_i^{G}, since by definition, $cost(\phi_i^{\mathsf{G}})$ only counts the $\vee$ and $\wedge$ operations of ϕ_i^{G}. The costs of j' are already included in C_g, so they do not have to be computed again. Without merging the repeating subexpressions that may occur in different ϕ_i^{G}, the cost of G_k is $\mathsf{C}_b = \sum_{j=1}^{|G|} cost(\phi_j^{\mathsf{G}}) \cdot b_j$.

In total, we get the cost $\mathsf{C}_g + \mathsf{C}_{oc1} + \mathsf{C}_{oc2} + \mathsf{C}_b$. □

Correctness of Applying $T_{ILP}^{\leftarrow}$ to a Greedy Algorithm Solution (Theorem 6.3)

We show that we can use Algorithm 2 for constructing the optimized circuit from the set of cliques returned by a greedy algorithm.

By Lemma 6.4, Algorithm 2 terminates. Let Cs be the set of cliques returned by the greedy algorithm. We now may reduce the cliques to an ILP solution as follows. First, sort each $C \in Cs$ by the gate indices, so that the first index is the smallest one. For all $C \in Cs$, fix $j = C[0]$ (the first element of C, i.e. its gate index is minimal), and assign $g_i^j = 1$ for all $i \in C$. We show that this assignment satisfies basic ILP constraints.

- In Algorithm 2, the gates are first of all sorted by types, and only the gates of the same type are fused. This satisfies the constraint (3) of ILP.
- As soon as a gate has been taken into a clique, this is not added to any other clique. If any gate is left, it is treated as a singleton clique. This satisfies the constraint (2).
- We have taken $g_i^j = 1$ for $j = C[0]$, so j is the unique clique representative, and (4) is satisfied.
- Algorithm 2 stops when it reaches gates of cost 0 and puts all of them into separate cliques, so that (5) is satisfied.

- Whatever greedy strategy we take, each of them accepts a clique iff the function goodClique returns true. For any gates i, j that belong to the same clique C, $\mathsf{fusable}^{\mathsf{G}}(i, j)$ holds. This satisfies the constraint (1) of ILP. In addition, goodClique assigns levels to the gates, which can be put into one-to-one correspondence with the variables ℓ_i of the constraint (6). For $\mathsf{pred}^{\mathsf{G}}(i, k)$, we have $\mathsf{level}(i) > \mathsf{level}(k)$, and for $\mathsf{cpred}^{\mathsf{G}}(i, k)$ we have $\mathsf{level}(i) > \mathsf{level}(k)$ iff the size of the clique to which i belongs is at least 2. The function goodClique checks it on the fly, and since if a clique has already been accepted, it will never be updated anymore, the condition will not be broken.

As has been shown in the feasibility proof of Section 6.5.6, the extended ILP constraints introduce new variables that make them satisfiable in any case. Hence we do not need to prove separately that all of them hold. □

6.6 Implementation and Evaluation

We have implemented the transformation of the program to a circuit, the optimization algorithms, and the transformation of the circuit according to the obtained set of cliques in SWI-Prolog [103]. The ILP is solved externally by the GLPK solver [44]. The optimized circuit is translated to a Sharemind program for evaluation.

The optimization algorithms have been tested on small programs. Since we are dealing with a relatively new problem, there are no good test sets, and we had to invent some sample programs ourselves. In general, the programs with private conditionals are related to evaluation of decision diagrams with private decisions. We provide five different programs, each with its own specificity. Their pseudocodes are given in Appendix A.

- loan (31 gates): A simple binary decision tree, which decides whether a person should be given a loan, based on its background data. Such simple applications are often used as an introduction to the decision tree topic. Only the leaves contain assignments, and the optimization is only trying to fuse the comparison operations that make the decisions. Uses only integer operations.

- sqrt (123 gates): Uses binary search to compute the square root of an integer. Since the input is private, it makes a fixed number of iterations. The division by 2 is on purpose inserted into both branches, modified in such a way that it cannot be trivially outlined without arithmetic theory. The optimizer does this outlining by fusing the divisions. Uses only integer operations.

- driver (53 gates): We took the decision tree that is applied to certain parameters of a piece of music in order to check how well it wakes up a sleepy car driver [74], assuming that the parameters used in this task are private. In this tree, some decisions require more complex operations, such as logarithms and inverses (computing Shannon entropy), so it was interesting to try to fuse them. Works with floating point arithmetic [51].

- stats (68 gates): The motivation for the problem is that choosing a particular statistical test for the analysis may depend on the type of data (ordinal, binary) [76]. Here we assume that the decision bits (which analysis to choose) are already given, but are private. The complex computation starts in leaves, where a particular statistical test is chosen. It chooses among the Student t-test, the Wilcoxon test, the Welch test, and the χ^2 test, whose privacy-preserving implementations are taken from [13]. Works with floating point arithmetic.

- erf (335 gates): The program evaluates the error function of a floating point number, which is represented as a triple (sign, significand, exponent) of integers [51]. The implementation is taken from [60]. In this program, the method chosen to compute the answer depends on the range in which the private input is located, and this choice cannot be leaked.

All our programs are vectorized. We treat vector operations as single gates, so that optimizing 10^6 operations per gate would be feasible. For simplicity, we assumed that all vectors in the program have the same length. Fusing together vector operations of different length can be treated as a future work.

We ran the optimizer on a Lenovo X201i laptop with a 4-core Intel Core i3 2.4GHz processor and 4GB of RAM running Ubuntu 12.04. The execution times are given in the Tables 6.1- 6.5. The rows correspond to different subcircuit depths d, which are constructed as described in Section 6.4.4. We tried all possible depths, until it was not possible to increase the depth anymore since all the subgraphs would have become unique. The columns correspond to the different optimization techniques. The columns grd_1, grd_2, and grd_3 are the three different greedy strategies that are described in Section 6.4.5. The columns $\mathsf{lp}_{\mathsf{basic}}$ and $\mathsf{lp}_{\mathsf{ext}}$ correspond to the mixed integer programming approach of Section 6.4.6, using the basic and the extended constraints respectively. We write lp for $\mathsf{lp}_{\mathsf{basic}}$ and $\mathsf{lp}_{\mathsf{ext}}$ taken together.

We compiled the optimized graphs into programs, executed them on Sharemind (three servers on a local 1Gbps network; the speed of the network is the bottleneck in these tests) and measured their running time. The runtime benchmarks can be found in Appendix B. For driver, sqrt, and loan, we see that the optimized programs are clearly grouped by their running times. The differences inside a

Table 6.1: Optimization times (s), loan

d	grd_1	grd_2	grd_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.046	0.047	0.053	0.097	0.12
1	0.049	0.041	0.042	0.095	0.11
2	0.042	0.043	0.050	0.083	0.12

Table 6.2: Optimization times (s), sqrt

d	grd_1	grd_2	grd_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.52	0.55	0.54	0.78	1.3
1	0.55	0.53	0.54	0.64	0.76
2	0.56	0.53	0.52	0.61	0.65
3	0.54	0.50	0.57	0.58	0.60
4	0.55	0.51	0.51	0.62	0.59

Table 6.3: Optimization times (s), driver

d	grd_1	grd_2	grd_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.12	0.11	0.13	0.18	0.38
1	0.084	0.082	0.081	0.16	0.16
2	0.075	0.083	0.076	0.14	0.16
3	0.082	0.077	0.078	0.12	0.17

Table 6.4: Optimization times (s), stats

d	grd_1	grd_2	grd_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.19	0.18	0.19	0.29	16
1	0.12	0.12	0.12	0.17	0.26
2	0.10	0.10	0.11	0.15	0.23
3	0.10	0.11	0.10	0.14	0.22
4	0.10	0.11	0.11	0.14	0.21
5	0.11	0.10	0.11	0.16	0.22

Table 6.5: Optimization times (s), erf

d	grd_1	grd_2	grd_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	43	43	–	47	48
1	5.5	5.5	–	–	–
2	3.6	3.6	8.5	8.1	–
3	3.2	3.3	6.1	6.3	–
4	2.7	2.8	2.7	3.5	–
5	2.6	2.6	2.6	3.1	530
6	2.6	2.6	2.6	3.0	15
7	2.6	2.6	2.8	3.0	15

single group are insignificant, so we may treat the entire group as having the same cost. For stats and erf, we do not see such a partitioning. The results are too varying, and hence we cannot claim if the optimization was harmful or useful. Running stats on 10 inputs shows advantage of $\mathsf{lp}_{\mathsf{basic}}$and $\mathsf{lp}_{\mathsf{ext}}$, but it gets lost on 100 inputs. This most probably indicates that the advantage has come from fusing together non-vector operations which are less significant for larger inputs.

The summary of the results is given in Table 6.6. For each program, we give the runtime range of its optimized versions, the runtime of the non-optimized version, and which strategies have been the best and the worst. Here grd_1 is the strategy that chooses the largest clique first, and grd_2 fuses the gates pairwise (grd_3 is quite similar to grd_2, so we do not differentiate them).

Since the runtime depends also on the number of rounds that we did not optimize, our results are not good for small inputs. However, as the total amount of communication and computation increases, our optimized programs are becoming more advantageous. While greedy approaches may outperform ILP approaches for smaller inputs, ILP is more stable for large inputs.

In general, it is preferable not to merge the initial gates into subcircuits (take depth 0). The greedy strategies work quite well for the given programs, but their results are too unpredictable and can be very good as well as very bad. The results of ILP are in general better. In practice, it would be good to estimate

Table 6.6: Execution times

$n = 10$	driver	sqrt	loan	erf	stats
time (s)	0.156-0.193	0.071-0.077	0.012-0.015	0.085-0.121	1.70-1.75
w/o opt.	0.156	0.073	0.016	0.091	1.76
best strat.	grd_1	depth 0	depth 0	depth 1,5+	$\mathsf{lp}_{\mathsf{basic}}$, $\mathsf{lp}_{\mathsf{ext}}$
worst strat.	grd_1 dpt. 0	depth 1	grd_2 dpt. 0	depth 2-4	grd_1,grd_2
$n = 10^3$	driver	sqrt	loan	erf	
time (s)	0.588-0.809	0.249-0.291	0.032-0.041	0.275-0.334	
w/o opt.	0.705	0.283	0.051	0.316	
best strat.	lp, grd_1	depth 0	depth 0	depth 1,4+	
worst strat.	grd_1 dpt. 0	grd_1 depth 3	grd_2 dpt. 0	depth 2,3	
$n = 10^6$	driver	sqrt	loan	erf	stats, $n = 100$
time (s)	200-336	97-120	10-14	95-111	136-146
w/o opt.	256	121	19.5	108	148
best strat.	lp, grd_1	depth 0	depth 0	depth 1,4+	no preference
worst strat.	grd_1 dpt. 0	depth 2+	grd_2 dpt. 0	depth 2,3	no preference

the approximate runtime of the program before it is actually executed, so that we could take the best variant. Our optimization seems to be most useful for library functions, where several different optimized versions can be compiled and benchmarked before choosing the final one.

6.7 Discussion

We could as well apply the costs of the verification phase (see Chapter 4) for the circuit optimization. However, this may result in having different optimal solutions for different phases. Since there is a positive correlation between the costs of the execution and the verification phases, we propose that it is sufficient to optimize the cost of the execution phase only, since it is more important.

Alternatively, to optimize the verification phase, we could apply our optimization to the *local* circuits computed by the parties. For example, if the computation of some party depends on some of its private values, then the prover may compute only one branch in the execution phase, while the verifiers should compute all of them and perform the oblivious choice. There are some choices in Sharemind protocols, but all of them are related to choosing between x and some linear combination involving x, so there are no gates that could be fused. We will see if the further development of Sharemind (e.g. floating point protocols) brings us more interesting cases.

6.8 Summary

We have presented an optimization for programs written in an imperative language with private conditions, aimed to reduce the computational overhead caused by branching on private variables. The reduction and the optimization algorithms are not restricted to any specific secure computation platforms. We have optimized and benchmarked some programs on Sharemind.

Currently, we are using arithmetic blackbox operations as the gates of the circuit. We have chosen arithmetic black boxes as subcircuits, since then it should be relatively easy to transform programs without knowing how exactly the black-box operations are constructed (inside, they may actually be some asymmetric protocols that are not decomposable further). As a future work, we could try to decompose the operations as much as possible, getting an arithmetic (or a boolean) circuit, possibly allowing to fuse together some parts of different blackbox functions. Taking into account vectors of different lengths would be another useful improvement.

CONCLUSION

In this thesis, we have developed new methods for building efficient protocols that are secure against covert and active adversaries. Seeing that our provably secure constructions obviously leak information to honest parties, we have looked at the honest parties from another point of view, modifying the standard security model.

We have presented a verification phase that can be run after the execution of a secure multiparty computation protocol, allowing to detect every party that has cheated. This gives a generic method for turning passively secure protocols to covertly and actively secure, depending on the way in which the verification is applied. Our method does not introduce significant overheads, and it is optimized for three party computation over rings. We also have provided an alternative verification mechanism that works better for computation over finite fields.

We have studied the problem of leaking the private data of one honest party to another honest party. We have shown that it is possible to define a relatively simple adversarial model for this purpose, and proposed some generic methods of achieving security in this new model. We have checked the security of our own verification protocols in this model, found some related vulnerabilities, and fixed them.

Finally, we studied optimizations that could be applied in both the protocol execution phase and the verification phase. Our optimization is related to circuit gates whose outputs will not be needed in the particular execution, but where the necessity depends on a private condition, which is not allowed to affect the control flow. We have proposed an automatic program optimization that reduces such repeating computation without leaking the condition bit. The resulting circuit is as secure as the initial circuit. The optimization is quite general, and can be applied to various privacy-preserving platforms.

APPENDIX A

OPTIMIZED SAMPLE PROGRAMS

driver

```
def mean(x):
    return floatMult(floatSum(x), inv(length(x)));

def entropy (x):
    return floatNeg(floatSum(floatMult(x, ln(x))));

def divide (x1, x2):
    return floatMult (x1, inv(x2));

def AP (v):
    return mean (v);

def AD (d):
    return mean (d);

def PE (n, t):
    return entropy (divide (n, t));

def PIE (i, ti):
    return entropy (divide (i, ti));

def main:
    private v, d, n, i, t, ti;
    input v, d, n, i, t, ti;

    private ap := AP(v);
    private ad := AD(d);
    private pe := PE(n,t);
    private pie := PIE(i,ti);

    if ap <= 900:
        if ad <= 13:
```

```
            if pe <= -5:
                y := 6
            else:
                y := 7;
        else:
            if pie <= -6:
                y := 5
            else
                y := 9;
    else:
        if ad <= 21:
            if pe <= -3:
                y := 8
            else:
                y := 7;
        else:
            if pie <= -4:
                y := 4
            else:
                y := 3;
    return y;
```

sqrt

```
def main:
    private a, x, y, mid;
    private answer := -1;
    input a;
    x := 0;
    y := a;
    mid := a >> 1;
    for i in range(10):
        if (mid * mid > a):
            y := mid;
            mid := (x + mid) >> 1;
        else if (mid * mid < a):
            x := mid;
            mid := (mid + y) >> 1;
        answer := mid;
    return answer;
```

loan

```
def main:
    private age, num_of_parents, num_of_children,
            income, answer;
    input age, num_of_parents, num_of_children,
            income;

    if age < 18:
```

```
        answer := 0
    else if age < 65:
        if num_of_children == 0:
            if income > 20:
                answer := 1
            else:
                answer := 0
        else if num_of_parents == 1:
            if income > 25:
                answer := 1
            else:
                answer := 0
        else:
            if income > 30:
                answer := 1
            else:
                answer := 0
    else if income > 40:
        answer := 1
    else:
        answer := 0;
    return answer;
```

stats

```
def mean (x):
    return floatMult (floatSum(x), inv(length(x)));

def variance (x):
    private w := floatMult (floatSquare (floatSum(x));
    private z1 := floatNeg (w, inv(n)));
    private z2 := floatSum (floatSquare(x));
    return floatMult (floatAdd (z1,z2), inv(n - 1));

def sdev (x):
    return sqrt (variance (x));

def sdev (x, y):
    public nxy := length(x) + length(y) - 2;
    public nx := length(x) - 1;
    public ny := length(y) - 1;
    private vx := floatMult (variance (x), nx);
    private vy := floatMult (variance (y), ny);
    return sqrt (floatMult (floatAdd (vx,vy), inv(nxy)));

def studenttest (x, y):
    private mx := mean (x);
    private my := mean (y);
    private z1 := floatAdd (mx, floatNeg(my));
```

```
    public nx := inv(length(x));
    public ny := inv(length(y));
    private sxy := sdev (x, y);
    public w := sqrt (floatAdd (n1, n2));
    private z2 := floatMult (sxy, w);

    return floatMult (z1,inv (z2));

def welchtest (x, y):
    private mx := mean (x);
    private my := mean (y);
    private z1 := floatAdd (mx, floatNeg(my));

    public nx := inv(length(x));
    public ny := inv(length(y));
    private vx := floatMult (variance (x), nx);
    private vy := floatMult (variance (y), ny);
    private z2 := sqrt (floatAdd (vx,vy));

    return floatMult (z1, inv(z2));

def wilcoxontest (x, y):
    private d := floatAdd (x, floatNeg(y));
    private s := floatSign (d);
    private dpr := floatAbs (d);

    s := sort (dpr, s);
    private r := rank0 (s);
    return floatSum (floatMult (s, r));

def contingencytable (x, y, c):
    private ct_x := floatOuterEqualitySums (x,c);
    private ct_y := floatOuterEqualitySums (y,c);
    return (ct_x, ct_y);

def chisquaretest (x, y, c);
    private ninv := inv (length(x));

    private (ct_x,ct_y) := contingencytable (x, y, c);
    private rx := floatSum (ct_x);
    private ry := floatSum (ct_y);
    private p  := floatAdd (ct_x,ct_y);

    private ex := floatMult (floatMult(p,rx), ninv);
    private ey := floatMult (floatMult(p,ry), ninv);

    private nex := floatNeg(ex);
    private ney := floatNeg(ey);
    private z1 := floatSquare (floatAdd (ct_x, nex));
    private z2 := floatSquare (floatAdd (ct_y, ney));
```

```
    private w1 := floatSum (floatMult (z1,inv(ex)));
    private w2 := floatSum (floatMult (z2,inv(ey)));
    return floatAdd (w1,w2);

def main:
    private result;
    private b1;  #is the distribution normal
    private b2;  #are the stdev and the mean known
    private b3;  #is it ordinal data

    input b1, b2, b3;

    # The first dataset
    private x;
    input x;

    # The second dataset
    private y;
    input y;

    # The set of possible classes for chi-squared test
    private c;
    input c;

    if b1 == 1:
        if b2 == 1:
            result := studenttest (x, y);
        else:
            result := welchtest (x, y);
    else:
        if b3 == 1:
            result := wilcoxontest (x, y);
        else:
            result := chisquaretest (x, y, c);

    return result;
```

erf

```
#fixpoint to floating point
def fix_to_float (y,t,n,q):
    private u := getbit (y, t);
    private s := 1;
    private e;
    private f;
    if u == 1:
        e := t + q + 1;
        f := y * (1 << (n-t-1));
    else
```

```
        e := t + q;
        f := y * (1 << (n-t));
    return (s,e,f);

#Multiply two floating point numbers
def float_mult (s1,e1,f1,s2,e2,f2,n):

    private lambda;
    s := (s1 == s2);
    e := e1 + e2;
    f := f1 * f2;
    #here --> and <-- are ring conversion operations
    f := ((f1 --> (2*n))*(f2 --> (2*n))) <-- n;
    lambda := f >> (n-1);
    if lambda == 0:
        f := f << 1;
        e := e - 1;
    return (s,e,f);

#Evaluate a polynomial of degree <= 12
def eval (x0,s,c):

    private x[13];
    x[0] := 1;
    x[1] := x0;
    x[2] := x0 * x0;
    x[3] := x[2] * x[1];
    x[4] := x[2] * x[2];
    x[5] := x[4] * x[1];
    x[6] := x[4] * x[2];
    x[7] := x[4] * x[3];
    x[8] := x[4] * x[4];
    x[9] := x[8] * x[1];
    x[10] := x[9] * x[2];
    x[11] := x[8] * x[3];
    x[12] := x[8] * x[4];

    private z1[13] := 0;
    private z2[13] := 0;

    for i in range(13):
        if s[i] == 1:
           z1[i] := z1[i] + c[i] * x[i];
        else if s[i] == -1:
           z2[i] := z2[i] + c[i] * x[i];

    return z1 - z2;


def gaussian_poly_0 (x):
```

```
    return eval (x,[0,1,-1,-1,-1,1,-1,-1,-1,1,-1,-1,1],
    [0, 37862129, 89, 12620065, 3115, 3797002, 27323,
         850652, 68415, 238867, 35736, 22843, 6588]);

def gaussian_poly_1 (x):
    return eval (x,[1,1,1,-1,1,-1,-1,1,1,1,1,1,1],
    [945472, 31405311, 18236798, 40079935, 23153761,
                5984925, 599861, 0, 0, 0, 0, 0, 0]);

def gaussian_poly_2 (x):
    return eval (x,[-1,1,-1,1,1,1,-1,1,1,1,1,1,1],
    [31613609, 134982639, 119986495, 59088711, 17266836,
               2930966, 247133, 3236, 636, 0, 0, 0, 0]);

def gaussian_poly_3 (x):
    return eval (x,[1,1,-1,1,-1,1,-1,1,1,1,1,1,1],
    [28778930, 7535740, 4967310, 1750656, 347929,
                 36972, 1641, 0, 0, 0, 0, 0, 0]);

def main:
    #the input
    private s, e, f;
    input s, e, f;

    public q := (1 << 14) - 1;
    public n := 32;
    public m := 25;

    public shift0 := n - m + 0 - 2;
    public shift1 := n - m + 1 - 2;
    public shift2 := n - m + 2 - 2;
    public shift3 := n - m + 3 - 2;
    public shift4 := n - m + 4 - 2;

    private f0 := f >> shift0;
    private f1 := f >> shift1;
    private f2 := f >> shift2;
    private f3 := f >> shift3;
    private f4 := f >> shift4;

    private g0, g_1, g_1_0, g_1_1;

    g0 := gaussian_poly_1 (f0);
    g_1_0 := gaussian_poly_2 (f0);
    g_1_1 := gaussian_poly_3 (f0);

    private u := f <- m;
    if u == 1:
        g_1 := g_1_1
    else
```

```
    g_1 := g_1_0;

public t_1 := 0;
public t0 := 0;
public t1 := 2 - 1;
public t2 := 2 - 2;
public t3 := 2 - 3;
public t4 := 2 - 4;

if e <= q - 4:
    e := q - 4;
if q + 3 <= e:
    e := q + 3;

private s_pr, e_pr, f_pr;
private index := e - q;

if index == -2:
    (s_pr,e_pr,f_pr) := float_mult(1,25,21361415,s,e,f,n);
else if index == -1:
    (s_pr,e_pr,f_pr) := fix_to_float (g_1,t_1,n,q);
else if index == 0:
    (s_pr,e_pr,f_pr) := fix_to_float (g0,t0,n,q);
else if index == 1:
    private g1 := gaussian_poly_0 (f1);
    (s_pr,e_pr,f_pr) := fix_to_float (g1,t1,n,q);
else if index == 2:
    private g2 := gaussian_poly_0 (f2);
    (s_pr,e_pr,f_pr) := fix_to_float (g2,t2,n,q);
else if index == 3:
    private g3 := gaussian_poly_0 (f3);
    (s_pr,e_pr,f_pr) := fix_to_float (g3,t3,n,q);
else if index == 4:
    private g4 := gaussian_poly_0 (f4);
    (s_pr,e_pr,f_pr) := fix_to_float (g4,t4,n,q);
else
    (s_pr,e_pr,f_pr) :=  (1, 0, 1);

return (s_pr, e_pr, f_pr);
```

APPENDIX B

RUNNING TIMES OF PROGRAMS AFTER OPTIMIZATION

The runtime benchmarks are given in Figures B.2-B.5. The X-axis (vertical) corresponds to different optimizations, including all the combinations of the 5 strategies with all used subcircuit depths. The Y-axis (horizontal) represents the running times. The parameter n is the vector length — the number of executions run in parallel.

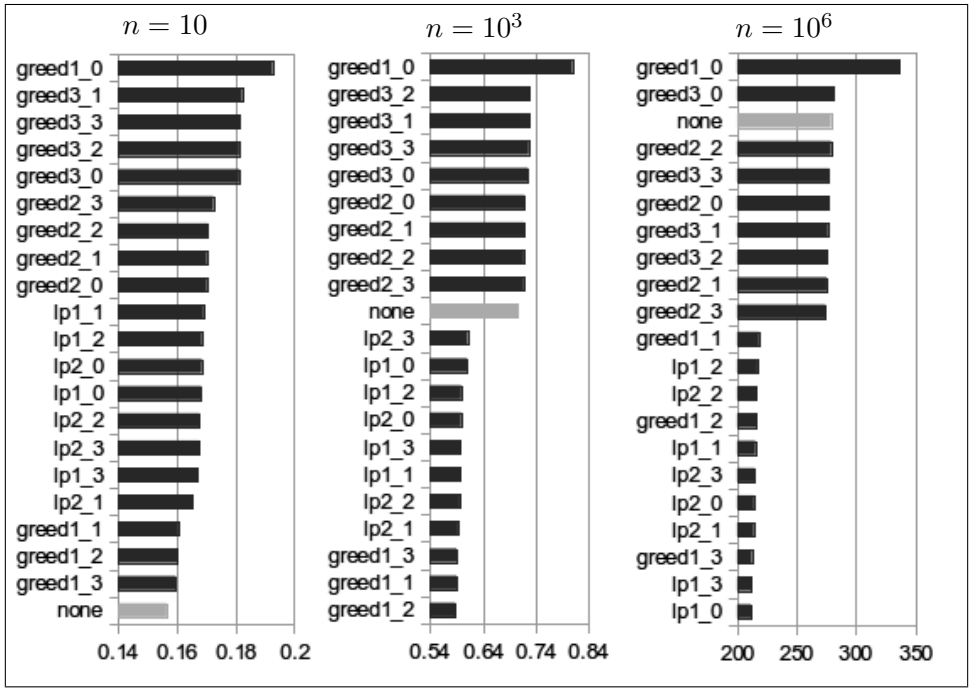


Figure B.1: Running times in seconds for driver

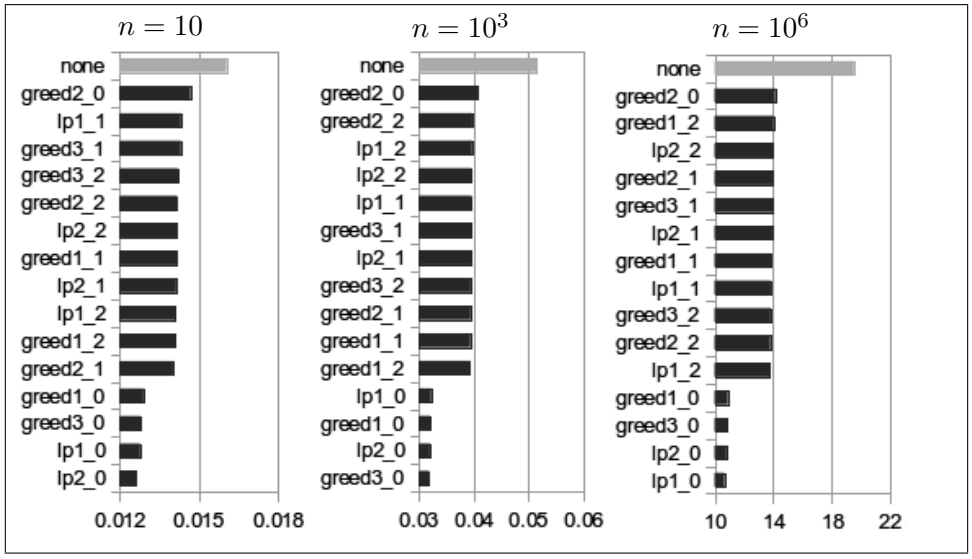


Figure B.2: Running times in seconds for loan

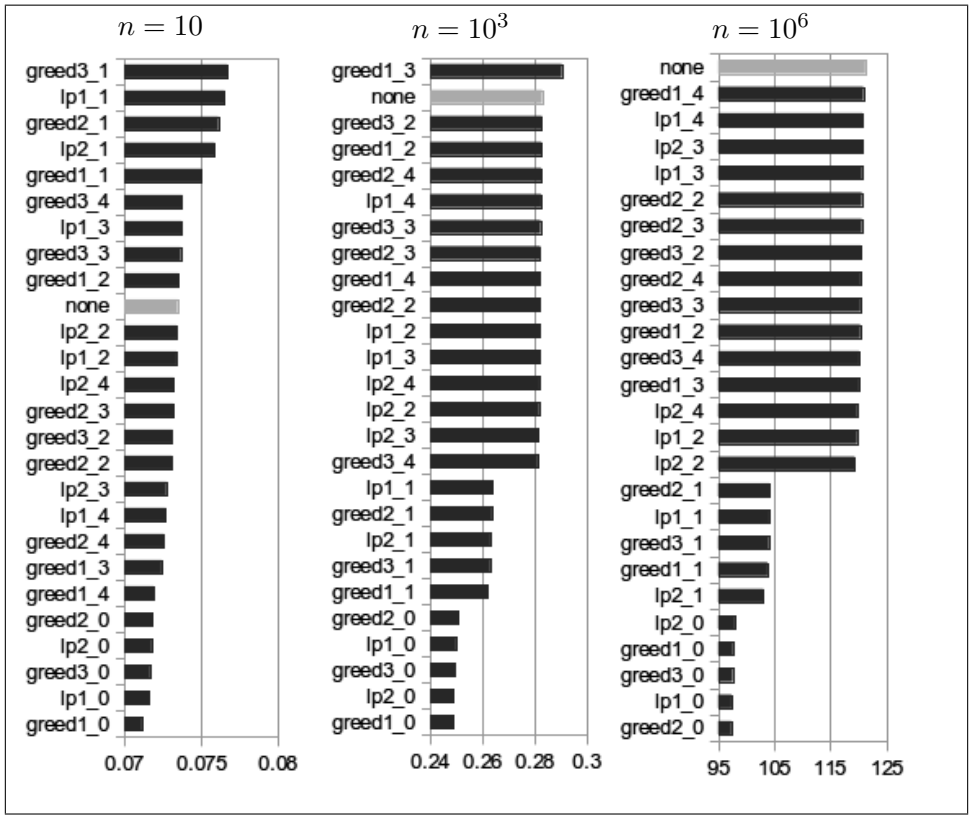


Figure B.3: Running times in seconds for sqrt

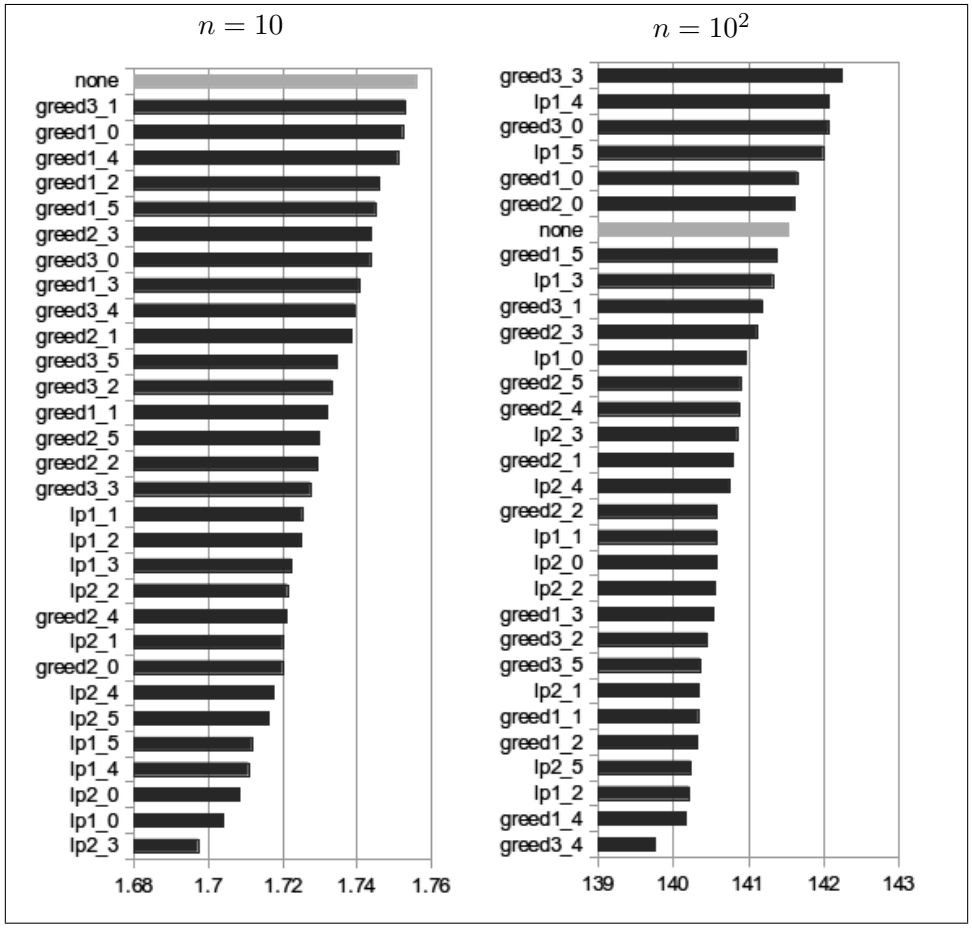


Figure B.4: Running times in seconds for stats

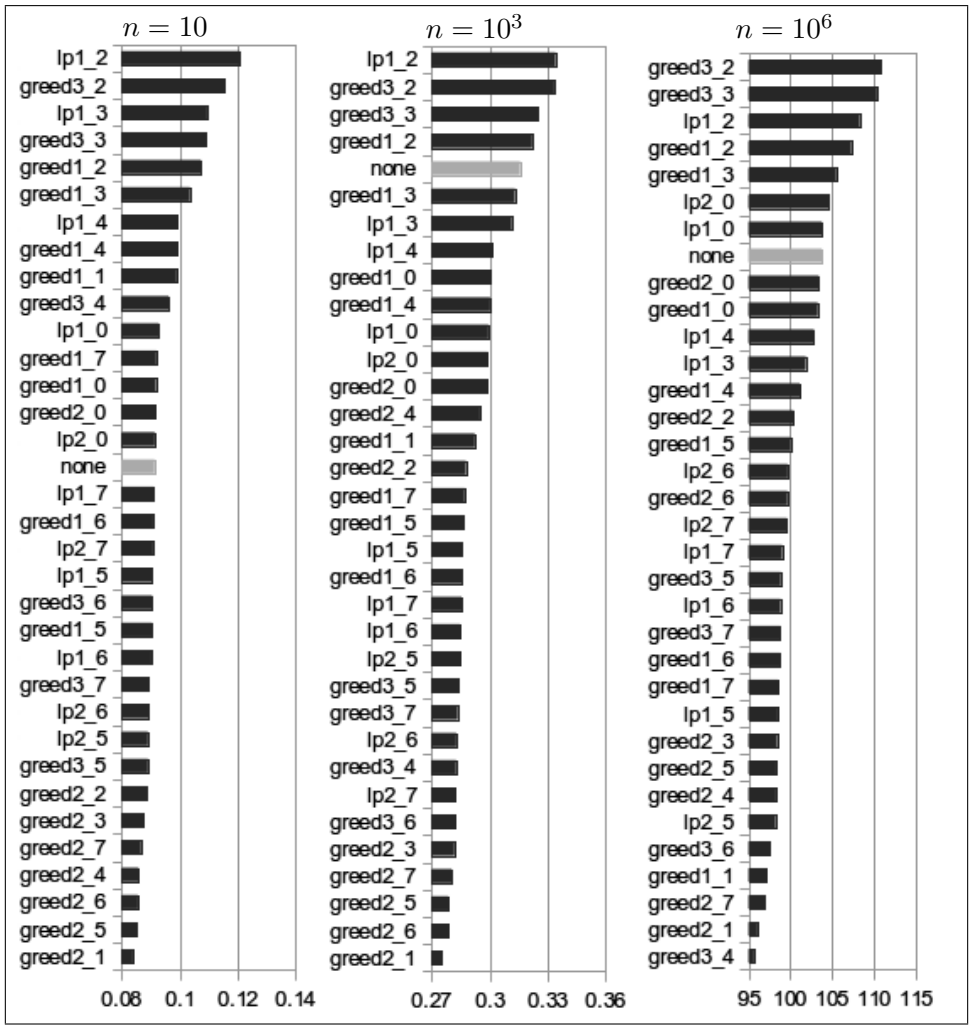


Figure B.5: Running times in seconds for erf

Bibliography

[1] Alwen, J., Katz, J., Maurer, U., Zikas, V.: Collusion-preserving computation. In: Safavi-Naini and Canetti [94], pp. 124–143, http://dx.doi.org/10.1007/978-3-642-32009-5_9

[2] Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. vol. 13 (2013)

[3] Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 805–817. CCS '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2976749.2978331

[4] Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient protocols for realistic adversaries. J. Cryptology 23(2), 281–343 (2010)

[5] Baum, C., Damgård, I., Orlandi, C.: Publicly auditable secure multi-party computation. In: Abdalla, M., Prisco, R.D. (eds.) Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings. LNCS, vol. 8642, pp. 175–196. Springer (2014), http://dx.doi.org/10.1007/978-3-319-10879-7_11

[6] Baum, C., Damgård, I., Toft, T., Zakarias, R.: Better preprocessing for secure multiparty computation. In: Manulis, M., Sadeghi, A.R., Schneider, S. (eds.) Applied Cryptography and Network Security: 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings. pp. 327–345. Springer International Publishing (2016)

[7] Baum, C., Orsini, E., Scholl, P.: Efficient secure multiparty computation with identifiable abort. In: Hirt, M., Smith, A.D. (eds.) Theory

of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I. LNCS, vol. 9985, pp. 461–490 (2016), http://dx.doi.org/10.1007/978-3-662-53641-4_18

[8] Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO. LNCS, vol. 576, pp. 420–432. Springer (1991)

[9] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: Snarks for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II. LNCS, vol. 8043, pp. 90–108. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40084-1_6

[10] Bitansky, N., Chiesa, A., Ishai, Y., Ostrovsky, R., Paneth, O.: Succinct non-interactive arguments via linear interactive proofs. In: Sahai [95], pp. 315–333, http://dx.doi.org/10.1007/978-3-642-36594-2_18

[11] Bogdanov, D., Jõemets, M., Siim, S., Vaht, M.: How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In: Böhme, R., Okamoto, T. (eds.) Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015. LNCS, vol. 8975, pp. 227–234. Springer (2015), http://dx.doi.org/10.1007/978-3-662-47854-7_14

[12] Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., Talviste, R.: Students and taxes: a privacy-preserving study using secure computation. PoPETs 2016(3), 117–135 (2016), http://www.degruyter.com/view/j/popets.2016.2016.issue-3/popets-2016-0019/popets-2016-0019.xml

[13] Bogdanov, D., Kamm, L., Laur, S., Sokk, V.: Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512 (2014), http://eprint.iacr.org/2014/512

[14] Bogdanov, D., Laud, P., Laur, S., Pullonen, P.: From Input Private to Universally Composable Secure Multi-party Computation Primitives. In: Datta, A., Fournet, C. (eds.) IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014. pp. 184–198. IEEE (2014), http://dx.doi.org/10.1109/CSF.2014.21

[15] Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: Russo, A., Tripp, O. (eds.) Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014. p. 53. ACM (2014), http://doi.acm.org/10.1145/2637113.2637119

[16] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.) ESORICS. LNCS, vol. 5283, pp. 192–206. Springer (2008)

[17] Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. Int. J. Inf. Sec. 11(6), 403–418 (2012)

[18] Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis - (short paper). In: Keromytis, A.D. (ed.) Financial Cryptography. LNCS, vol. 7397, pp. 57–64. Springer (2012)

[19] Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers. LNCS, vol. 5628, pp. 325–343. Springer (2009), http://dx.doi.org/10.1007/978-3-642-03549-4_20

[20] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012. pp. 309–325. ACM (2012), http://doi.acm.org/10.1145/2090236.2090262

[21] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS. pp. 136–145. IEEE Computer Society (2001)

[22] Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings. LNCS, vol. 4392, pp. 61–85. Springer (2007), http://dx.doi.org/10.1007/978-3-540-70936-7_4

[23] Canetti, R., Vald, M.: Universally composable security with local adversaries. In: Visconti and Prisco [102], pp. 281–301, http://dx.doi.org/10.1007/978-3-642-32928-9_16

[24] Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., Prisco, R.D. (eds.) Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings. LNCS, vol. 6280, pp. 182–199. Springer (2010), http://dx.doi.org/10.1007/978-3-642-15317-4_13

[25] Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings. LNCS, vol. 3378, pp. 342–362. Springer (2005), http://dx.doi.org/10.1007/978-3-540-30576-7_19

[26] Cramer, R., Damgård, I., Maurer, U.M.: General secure multi-party computation from any linear secret-sharing scheme. In: Preneel, B. (ed.) EUROCRYPT. LNCS, vol. 1807, pp. 316–334. Springer (2000)

[27] Cunningham, R., Fuller, B., Yakoubov, S.: Catching MPC cheaters: Identification and openability. Cryptology ePrint Archive, Report 2016/611 (2016), http://eprint.iacr.org/2016/611

[28] Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Jarecki and Tsudik [50], pp. 160–179, http://dx.doi.org/10.1007/978-3-642-00468-1_10

[29] Damgård, I., Geisler, M., Nielsen, J.B.: From passive to covert security at low cost. In: Micciancio, D. (ed.) TCC. LNCS, vol. 5978, pp. 128–145. Springer (2010)

[30] Damgård, I., Keller, M., Larraia, E., Miles, C., Smart, N.P.: Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In: Visconti and Prisco [102], pp. 241–263, http://dx.doi.org/10.1007/978-3-642-32928-9_14

[31] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS. LNCS, vol. 8134, pp. 1–18. Springer (2013)

[32] Damgård, I., Nielsen, J.B., Nielsen, M., Ranellucci, S.: Gate-scrambling revisited - or: The tinytable protocol for 2-party secure computation. Cryptology ePrint Archive, Report 2016/695 (2016), http://eprint.iacr.org/2016/695

[33] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini and Canetti [94], pp. 643–662

[34] Damgård, I., Toft, T., Zakarias, R.W.: Fast multiparty multiplications from shared bits. Cryptology ePrint Archive, Report 2016/109 (2016), http://eprint.iacr.org/

[35] Damgård, I., Zakarias, S.: Constant-overhead secure computation of boolean circuits using preprocessing. In: Sahai [95], pp. 621–641, http://dx.doi.org/10.1007/978-3-642-36594-2_35

[36] Damiani, E., Bellandi, V., Cimato, S., Gianini, G., Spindler, G., Grenzer, M., Heitmüller, N., Schmechel, P.: PRACTICE Deliverable D31.2: risk-aware deployment and intermediate report on status of legislative developments in data protection (October 2015), available from http://www.practice-project.eu

[37] Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S.: Automated synthesis of optimized circuits for secure computation. In: Ray et al. [93], pp. 1504–1517, http://doi.acm.org/10.1145/2810103.2813678

[38] Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014. The Internet Society (2015), http://www.internetsociety.org/events/ndss-symposium-2015

[39] Franz, M., Holzer, A., Katzenbeisser, S., Schallhart, C., Veith, H.: CBMC-GC: an ANSI C compiler for secure two-party computations. In: Cohen, A. (ed.) Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. LNCS, vol. 8409, pp. 244–249. Springer (2014), http://dx.doi.org/10.1007/978-3-642-54807-9_15

[40] Frederiksen, T.K., Keller, M., Orsini, E., Scholl, P.: A Unified Approach to MPC with Preprocessing Using OT. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I. LNCS, vol. 9452, pp. 711–735. Springer (2015), http://dx.doi.org/10.1007/978-3-662-48797-6_29

[41] Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. Cryptology ePrint Archive, Report 2016/944 (2016), http://eprint.iacr.org/2016/944

[42] Galil, Z., Haber, S., Yung, M.: Cryptographic computation: Secure fault-tolerant protocols and the public-key model (extended abstract). In: Pomerance, C. (ed.) Advances in Cryptology - CRYPTO 87, LNCS, vol. 293, pp. 135–155. Springer Berlin Heidelberg (1988), http://dx.doi.org/10.1007/3-540-48184-2_10

[43] Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO. LNCS, vol. 6223, pp. 465–482. Springer (2010)

[44] GLPK: GNU Linear Programming Kit. http://www.gnu.org/software/glpk

[45] Goldreich, O.: Foundations of Cryptography: Basic Tools, vol. 1. Cambridge University Press, New York, NY, USA (2000)

[46] Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, New York, NY, USA (2004)

[47] Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: Aho, A.V. (ed.) STOC. pp. 218–229. ACM (1987)

[48] Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., del Cuvillo, J.: Using innovative instructions to create trustworthy software solutions. In: Lee and Shi [69], p. 11, http://doi.acm.org/10.1145/2487726.2488370

[49] Ishai, Y., Ostrovsky, R., Zikas, V.: Secure multi-party computation with identifiable abort. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa

Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II. LNCS, vol. 8617, pp. 369–386. Springer (2014), http://dx.doi.org/10.1007/978-3-662-44381-1_21

[50] Jarecki, S., Tsudik, G. (eds.): Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5443. Springer (2009), http://dx.doi.org/10.1007/978-3-642-00468-1

[51] Kamm, L., Willemson, J.: Secure floating point arithmetic and private satellite collision analysis. Int. J. Inf. Secur. 14(6), 531–548 (Nov 2015), http://dx.doi.org/10.1007/s10207-014-0271-8

[52] Katz, J., Lindell, Y.: Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series). Chapman & Hall/CRC (2007)

[53] Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: Sahai [95], pp. 477–498, http://dx.doi.org/10.1007/978-3-642-36594-2_27

[54] Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 830–842. ACM (2016), http://doi.acm.org/10.1145/2976749.2978357

[55] Kennedy, W.S., Kolesnikov, V., Wilfong, G.: Overlaying circuit clauses for secure computation. Cryptology ePrint Archive, Report 2016/685 (2016), http://eprint.iacr.org/2016/685

[56] Kerik, L., Laud, P., Randmets, J.: Optimizing MPC for robust and scalable integer and floating-point arithmetic. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. LNCS, vol. 9604, pp. 271–287. Springer (2016), http://dx.doi.org/10.1007/978-3-662-53357-4_18

[57] Kolesnikov, V., Malozemoff, A.J.: Public verifiability in the covert model (almost) for free. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory

and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II. LNCS, vol. 9453, pp. 210–235. Springer (2015), http://dx.doi.org/10.1007/978-3-662-48800-3_9

[58] Kreuter, B., Shelat, A., Shen, C.: Billion-gate secure computation with malicious adversaries. In: Kohno, T. (ed.) Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012. pp. 285–300. USENIX Association (2012), https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kreuter

[59] Krips, T., Willemson, J.: Hybrid model of fixed and floating point numbers in secure multiparty computations. In: Chow, S.S.M., Camenisch, J., Hui, L.C.K., Yiu, S. (eds.) Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings. LNCS, vol. 8783, pp. 179–197. Springer (2014), http://dx.doi.org/10.1007/978-3-319-13257-0_11

[60] Krips, T., Willemson, J.: Hybrid model of fixed and floating point numbers in secure multiparty computations. In: International Conference on Information Security. pp. 179–197. Springer (2014)

[61] Lapets, A., Volgushev, N., Bestavros, A., Jansen, F., Varia, M.: Secure Multi-Party Computation for Analytics Deployed as a Lightweight Web Application (July 2016), http://www.cs.bu.edu/techreports/pdf/2016-008-mpc-lightweight-web-app.pdf, cS Dept., Boston University, Tech.Rep.BUCS-TR-2016-008

[62] Laud, P., Pankova, A.: Verifiable Computation in Multiparty Protocols with Honest Majority. In: Chow, S.S.M., Liu, J.K., Hui, L.C.K., Yiu, S. (eds.) Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, China, October 9-10, 2014. Proceedings. LNCS, vol. 8782, pp. 146–161. Springer (2014), http://dx.doi.org/10.1007/978-3-319-12475-9_11

[63] Laud, P., Pankova, A.: Preprocessing-based verification of multiparty protocols with honest majority. Cryptology ePrint Archive, Report 2015/674 (2015), http://eprint.iacr.org/

[64] Laud, P., Pankova, A.: Optimizing secure computation programs with private conditionals. In: Lam, K., Chi, C., Qing, S. (eds.) Information and Communications Security - 18th International Conference, ICICS

2016, Singapore, November 29 - December 2, 2016, Proceedings. LNCS, vol. 9977, pp. 418–430. Springer (2016), http://dx.doi.org/10.1007/978-3-319-50011-9_32

[65] Laud, P., Pankova, A.: Securing multiparty protocols against the exposure of data to honest parties. In: Livraga, G., Torra, V., Aldini, A., Martinelli, F., Suri, N. (eds.) Data Privacy Management and Security Assurance - 11th International Workshop, DPM 2016 and 5th International Workshop, QASA 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings. LNCS, vol. 9963, pp. 165–180. Springer (2016), http://dx.doi.org/10.1007/978-3-319-47072-6_11

[66] Laud, P., Pettai, M.: Secure multiparty sorting protocols with covert privacy. In: Brumley, B.B., Röning, J. (eds.) Secure IT Systems - 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings. LNCS, vol. 10014, pp. 216–231 (2016), http://dx.doi.org/10.1007/978-3-319-47560-8_14

[67] Laud, P., Randmets, J.: A domain-specific language for low-level secure multiparty computation protocols. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015. pp. 1492–1503. ACM (2015), http://doi.acm.org/10.1145/2810103.2813664

[68] Laur, S., Willemson, J., Zhang, B.: Round-Efficient Oblivious Database Manipulation. In: Proceedings of the 14th International Conference on Information Security. ISC'11. pp. 262–277 (2011)

[69] Lee, R.B., Shi, W. (eds.): HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013. ACM (2013), http://dl.acm.org/citation.cfm?id=2487726

[70] Lindell, Y.: Fast cut-and-choose based protocols for malicious and covert adversaries. Cryptology ePrint Archive, Report 2013/079 (2013), http://eprint.iacr.org/2013/079

[71] Lindell, Y., Riva, B.: Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries. In: Ray et al. [93], pp. 579–590, http://doi.acm.org/10.1145/2810103.2813666

[72] Lipmaa, H.: Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In: Sako, K., Sarkar, P. (eds.)

Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I. LNCS, vol. 8269, pp. 41–60. Springer (2013), http://dx.doi.org/10.1007/978-3-642-42033-7_3

[73] Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient ram-model secure computation. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 623–638 (2014), http://dx.doi.org/10.1109/SP.2014.46

[74] Liu, N.H., Chiang, C.Y., Hsu, H.M.: Improving driver alertness through music selection using a mobile eeg to detect brainwaves. Sensors 13(7), 8199–8221 (2013)

[75] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - a secure two-party computation system. In: Proceedings of the 13th USENIX Security Symposium. pp. 287–302. USENIX Association, Berkeley, CA, USA (2004)

[76] McCrum-Gardner, E.: Which is the correct statistical test to use? British Journal of Oral and Maxillofacial Surgery 46(1), 38–41 (2008)

[77] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Lee and Shi [69], p. 10, http://doi.acm.org/10.1145/2487726.2488368

[78] Merkle, R.C.: Secrecy, authentication, and public key systems. Ph.D. thesis, Stanford University (1979)

[79] Mitchell, J.C., Sharma, R., Stefan, D., Zimmerman, J.: Information-flow control for programming on encrypted data. In: Chong, S. (ed.) 25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012. pp. 45–60. IEEE Computer Society (2012), http://dx.doi.org/10.1109/CSF.2012.30

[80] Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: The garbled circuit approach. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. pp. 591–602. CCS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2810103.2813705

[81] Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, New York, NY, USA (1995)

[82] Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: Hicks, M.W. (ed.) Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007. pp. 21–30. ACM (2007), http://doi.acm.org/10.1145/1255329.1255333

[83] Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini and Canetti [94], pp. 681–700, http://dx.doi.org/10.1007/978-3-642-32009-5_40

[84] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT. pp. 223–238 (1999)

[85] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: IEEE Symposium on Security and Privacy. pp. 238–252. IEEE Computer Society (2013), http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6547086

[86] Pettai, M., Laud, P.: Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In: Fournet, C., Hicks, M.W., Viganò, L. (eds.) IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015. pp. 75–89. IEEE (2015), http://dx.doi.org/10.1109/CSF.2015.13

[87] Pikma, T.: Auditing of Secure Multiparty Computations. Master's thesis, Institute of Computer Science, University of Tartu (2014)

[88] Planul, J., Mitchell, J.C.: Oblivious program execution and path-sensitive non-interference. In: 2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013. pp. 66–80. IEEE (2013), http://dx.doi.org/10.1109/CSF.2013.12

[89] Pollard, J.M.: The fast fourier transform in a finite field. Mathematics of computation 25(114), 365–374 (1971)

[90] Pruulmann-Vengerfeldt, P., Kamm, L., Talviste, R., Laud, P., Bogdanov, D.: Capability Model, UaESMC deliverable 1.1 (March 2012), http://www.usable-security.eu/workpackages-and-reports/wp1-requirements-gathering/d11.html

[91] Pullonen, P.: Actively Secure Two-Party Computation: Efficient Beaver Triple Generation. Master's thesis, University of Tartu, Aalto University (2013)

[92] Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: A programming language for generic, mixed-mode multiparty computations. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 655–670 (2014), http://dx.doi.org/10.1109/SP.2014.48

[93] Ray, I., Li, N., Kruegel, C. (eds.): Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015. ACM (2015), http://dl.acm.org/citation.cfm?id=2810103

[94] Safavi-Naini, R., Canetti, R. (eds.): Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings, LNCS, vol. 7417. Springer (2012)

[95] Sahai, A. (ed.): Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings, LNCS, vol. 7785. Springer (2013), http://dx.doi.org/10.1007/978-3-642-36594-2

[96] Schneier, B.: Data is a toxic asset (March 2016), https://www.schneier.com/blog/archives/2016/03/data_is_a_toxic.html

[97] Schröpfer, A., Kerschbaum, F., Müller, G.: L1 - an intermediate language for mixed-protocol secure computation. In: Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2011, Munich, Germany, 18-22 July 2011. pp. 298–307. IEEE Computer Society (2011), http://dx.doi.org/10.1109/COMPSAC.2011.46

[98] Setty, S.T.V., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking proof-based verified computation a few steps closer to practicality. In: USENIX Security Symposium (2012)

[99] Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979), http://doi.acm.org/10.1145/359168.359176

[100] Spini, G., Fehr, S.: Cheater detection in SPDZ multiparty computation. In: Nascimento, A.C.A., Barreto, P. (eds.) Information Theoretic Security - 9th International Conference, ICITS 2016, Tacoma, WA, USA, August 9-12, 2016, Revised Selected Papers. LNCS, vol. 10015, pp. 151–176 (2016), http://dx.doi.org/10.1007/978-3-319-49175-2_8

[101] Vaht, M.: The Analysis and Design of a Privacy-Preserving Survey System. Master's thesis, Institute of Computer Science, University of Tartu (2015)

[102] Visconti, I., Prisco, R.D. (eds.): Security and Cryptography for Networks - 8th International Conference, SCN 2012, Amalfi, Italy, September 5-7, 2012. Proceedings, LNCS, vol. 7485. Springer (2012), http://dx.doi.org/10.1007/978-3-642-32928-9

[103] Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)

[104] Yao, A.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982. pp. 160–164. IEEE Computer Society (1982), http://dx.doi.org/10.1109/SFCS.1982.38

[105] Zahur, S., Evans, D.: Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153 (2015), http://eprint.iacr.org/2015/1153

[106] Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: Sadeghi, A., Gligor, V.D., Yung, M. (eds.) 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. pp. 813–826. ACM (2013), http://doi.acm.org/10.1145/2508859.2516752

ACKNOWLEDGMENTS

The main support of author's PhD studies was coming from the Software Technologies and Applications Competence Centre (STACC), the Estonian Research Council through project IUT27-1, and the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 284731 (UaESMC). For the international movements, the author has also received support from Information Technology Foundation for Education (HITSA) 7-3.2/78-14, and European Cooperation in Science and Technology Short-Term Scientific Mission COST-STSM-IC1306-30409. The author also recognizes the support from the European Regional Development Fund through the Estonian Center of Excellence in Computer Science (EXCS), and by the European Social Fund through the Estonian Doctoral School in Information and Communication Technology (IKTDK).

The author would like to to thank Roman Jagomägis who has implemented Sharemind auditability mechanism especially for this work, finishing the missing link between the execution and the verification phases, allowing to fully benchmark the obtained solution. Author's job would be much more difficult without handy protocol DSL of Jaak Randmets. In addition, the author thanks all the other Cybernetica AS workers who have provided the author with Sharemind, which was an excellent basis on which the experiments could be performed, verifying author's theoretical results.

The most significant support was coming from the supervisors, whom the author especially thanks. The main supervisor Peeter Laud was tortured by the author starting from the BSc time. The co-supervisor Sven Laur joined the process already after the PhD studies had been executed, but nevertheless provided some insightful thoughts and helped with the organisatorial part. The ex-supervisor Margus Niitsoo did not withstand until the end, but his support was essential during the author's transition between MSc and PhD studies.

The author also thanks Rasmus Erlemann, Prastudy Fauzi, Liina Kamm, Margus Niitsoo, Pille Pullonen, Yauhen Yakimenka, Bingsheng Zhang for assistance with proofreading of this thesis.

Finally, the author would like to thank Jan Willemson who shared with the author his wisdom of life, and who demonstrated the daily work of a real researcher.

KOKKUVÕTE
(SUMMARY IN ESTONIAN)

TÕHUS PEIT- JA AKTIIVSE RÜNDAJA VASTU KAITSTUD TURVALINE ÜHISARVUTUS

Turvaline ühisarvutus on tänapäevase krüptograafia üks tähtsamaid kasutusviise, mis kasutab elegantseid matemaatilisi lahendusi praktiliste rakenduste ehitamiseks. See tehnoloogia võimaldab mitmel erineval andmeomanikul teha oma andmetega suvalisi ühiseid arvutusi, ilma et nad neid andmeid üksteisele avaldaks.

Ühisarvutuse turvalisuse aluseks on sageli teatud eeldused osapoolte käitumise kohta. Passiivselt turvaline protokoll ei lekita andmeid seni, kuni kõik osapooled käituvad ausalt. Peitründaja vastu turvaline protokoll töötab eeldusel, et ükski osapool ei hakka käituma reeglitevastaselt juhul, kui teised osapooled võivad seda märgata. Aktiivselt turvaline protokoll talub osapoolte käitumist ükskõik millisel viisil.

Käesolevas töös pakutakse välja üldine meetod, mis teisendab passiivse ründaja vastu turvalised ühisarvutusprotokollid turvaliseks kas peit- või aktiivse ründaja vastu. Meetod on optimeeritud kolme osapoolega arvutusteks üle algebraliste ringide $\mathbb{Z}_{2^n}$; praktikas on see väga efektiivne mudel, milles realiseerida pärismaailma rakendusi.

Lisaks uurib käesolev töö rünnete uut eesmärki, mis seisneb mingi ausa osapoole vaate manipuleerimises sellisel viisil, et ta saaks midagi teada teise ausa osapoole privaatsete andmete kohta. Ründaja ise ei tarvitse seda infot üldse teada saada. Sellised ründed on olulised, sest need tekitavad ausale osapoolele kohustuse puhastada oma süsteem teiste osapoolte andmetest. See ülesanne võib olla vägagi mittetriviaalne.

Lõpuks uurib käesolev töö üleliigsete arvutuste probleemi tuvalise ühisarvu-

tuse rakendustes. Mõnedel juhtudel peavad osapooled otsustama, mis suunas peab nende arvutus edasi minema. Kui see otsus sõltub privaatsetest andmetest, ei tohi ükski osapool haru valikust midagi teada, nii et üldjuhul peavad osapooled läbi viima arvutused kõigis harudes. Harude suure arvu korral võib arvutuslik lisakulu olla ülisuur, sest enamik vahetulemustest visatakse ära. Käesolevas töös esitatakse optimeerimisviis, mis selliseid lisakulusid vähendab.

LIST OF ORIGINAL PUBLICATIONS

- Laud, P., Pankova, A.: Verifiable Computation in Multiparty Protocols with Honest Majority. In: Chow, S.S.M., Liu, J.K., Hui, L.C.K., Yiu, S. (eds.) Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, China, October 9-10, 2014. Proceedings. LNCS, vol. 8782, pp. 146–161. Springer (2014), http://dx.doi.org/10.1007/978-3-319-12475-9_11.

- Laud, P., Pankova, A.: Securing multiparty protocols against the exposure of data to honest parties. In: Livraga, G., Torra, V., Aldini, A., Martinelli, F., Suri, N. (eds.) Data Privacy Management and Security Assurance - 11th International Workshop, DPM 2016 and 5th International Workshop, QASA 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings. LNCS, vol. 9963, pp. 165–180. Springer (2016), http://dx.doi.org/10.1007/978-3-319-47072-6_11.

- Laud, P., Pankova, A.: Optimizing secure computation programs with private conditionals. In: Lam, K., Chi, C., Qing, S. (eds.) Information and Communications Security - 18th International Conference, ICICS 2016, Singapore, November 29 - December 2, 2016, Proceedings. LNCS, vol. 9977, pp. 418–430. Springer (2016), http://dx.doi.org/10.1007/978-3-319-50011-9_32.

UNPUBLISHED RESULTS USED IN THIS THESIS

- Laud, P., Pankova, A.: Preprocessing-based verification of multiparty protocols with honest majority. Cryptology ePrint Archive, Report 2015/674 (2015), http://eprint.iacr.org/.

CURRICULUM VITAE

Personal data

Name	Alisa Pankova
Birth	October 20th, 1989, Tartu, Estonia
Citizenship	Estonian
Languages	English, Estonian, German, Japanese, Russian
E-mail	alisa.pankova@cyber.ee

Education

2013–	University of Tartu, Ph.D. candidate in Comp. Science
2011–2013	University of Tartu, M.Sc. in Computer Science
2008–2011	University of Tartu, B.Sc. in Computer Science

Employment

01.01.2016–	STACC, Junior Researcher (0,50)
01.01.2016–	Cybernetica AS, Junior Researcher (0,50)
01.09.2015–31.12.2015	STACC, Junior Researcher (0,20)
01.12.2013–31.12.2015	Cybernetica AS, Junior Researcher (0,20)
01.12.2013–31.08.2015	STACC, Junior Researcher (0,80)
01.02.2013–30.11.2013	STACC, Junior Researcher (1,00)
01.08.2012–31.01.2013	University of Tartu, Programming specialist (0,50)
01.07.2012–31.08.2012	Cybernetica AS, intern (1,00)
01.08.2011–31.07.2012	University of Tartu, Programming specialist (0,50)
01.07.2011–31.08.2011	Cybernetica AS, intern (1,00)

ELULOOKIRJELDUS

Isikuandmed

Nimi	Alisa Pankova
Sünniaeg ja -koht	20. oktoober 1989, Tartu, Eesti
Kodakondsus	eestlane
Keelteoskus	eesti, inglise, jaapani, saksa, vene
E-post	alisa.pankova@cyber.ee

Haridustee

2013–	Tartu Ülikool, informaatika doktorant
2011–2013	Tartu Ülikool, MSc informaatikas
2008–2011	Tartu Ülikool, BSc informaatikas

Teenistuskäik

01.01.2016–	STACC OÜ, Nooremteadur (0,50)
01.01.2016–	Cybernetica AS, Nooremteadur (0,50)
01.09.2015–31.12.2015	STACC OÜ, Nooremteadur (0,20)
01.12.2013–31.12.2015	Cybernetica AS, Nooremteadur (0,20)
01.12.2013–31.08.2015	STACC OÜ, Nooremteadur (0,80)
01.02.2013–30.11.2013	STACC OÜ, Nooremteadur (1,00)
01.08.2012–31.01.2013	Tartu Ülikool, Programmeerimise spetsialist (0,50)
01.07.2012–31.08.2012	Cybernetica AS, praktikant (1,00)
01.08.2011–31.07.2012	Tartu Ülikool, Programmeerimise spetsialist (0,50)
01.07.2011–31.08.2011	Cybernetica AS, praktikant (1,00)

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

1. **Mati Heinloo.** The design of nonhomogeneous spherical vessels, cylindrical tubes and circular discs. Tartu, 1991, 23 p.
2. **Boris Komrakov.** Primitive actions and the Sophus Lie problem. Tartu, 1991, 14 p.
3. **Jaak Heinloo.** Phenomenological (continuum) theory of turbulence. Tartu, 1992, 47 p.
4. **Ants Tauts.** Infinite formulae in intuitionistic logic of higher order. Tartu, 1992, 15 p.
5. **Tarmo Soomere.** Kinetic theory of Rossby waves. Tartu, 1992, 32 p.
6. **Jüri Majak.** Optimization of plastic axisymmetric plates and shells in the case of Von Mises yield condition. Tartu, 1992, 32 p.
7. **Ants Aasma.** Matrix transformations of summability and absolute summability fields of matrix methods. Tartu, 1993, 32 p.
8. **Helle Hein.** Optimization of plastic axisymmetric plates and shells with piece-wise constant thickness. Tartu, 1993, 28 p.
9. **Toomas Kiho.** Study of optimality of iterated Lavrentiev method and its generalizations. Tartu, 1994, 23 p.
10. **Arne Kokk.** Joint spectral theory and extension of non-trivial multiplicative linear functionals. Tartu, 1995, 165 p.
11. **Toomas Lepikult.** Automated calculation of dynamically loaded rigid-plastic structures. Tartu, 1995, 93 p, (in Russian).
12. **Sander Hannus.** Parametrical optimization of the plastic cylindrical shells by taking into account geometrical and physical nonlinearities. Tartu, 1995, 74 p, (in Russian).
13. **Sergei Tupailo.** Hilbert's epsilon-symbol in predicative subsystems of analysis. Tartu, 1996, 134 p.
14. **Enno Saks.** Analysis and optimization of elastic-plastic shafts in torsion. Tartu, 1996, 96 p.
15. **Valdis Laan.** Pullbacks and flatness properties of acts. Tartu, 1999, 90 p.
16. **Märt Põldvere.** Subspaces of Banach spaces having Phelps' uniqueness property. Tartu, 1999, 74 p.
17. **Jelena Ausekle.** Compactness of operators in Lorentz and Orlicz sequence spaces. Tartu, 1999, 72 p.
18. **Krista Fischer.** Structural mean models for analyzing the effect of compliance in clinical trials. Tartu, 1999, 124 p.
19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
20. **Jüri Lember.** Consistency of empirical k-centres. Tartu, 1999, 148 p.
21. **Ella Puman.** Optimization of plastic conical shells. Tartu, 2000, 102 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.

23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
25. **Maria Zeltser.** Investigation of double sequence spaces by soft and hard analitical methods. Tartu, 2001, 154 p.
26. **Ernst Tungel.** Optimization of plastic spherical shells. Tartu, 2001, 90 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 p.
28. **Rainis Haller.** *M(r,s)*-inequalities. Tartu, 2002, 78 p.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
30. Töö kaitsmata.
31. **Mart Abel.** Structure of Gelfand-Mazur algebras. Tartu, 2003. 94 p.
32. **Vladimir Kuchmei.** Affine completeness of some ockham algebras. Tartu, 2003. 100 p.
33. **Olga Dunajeva.** Asymptotic matrix methods in statistical inference problems. Tartu 2003. 78 p.
34. **Mare Tarang.** Stability of the spline collocation method for volterra integro-differential equations. Tartu 2004. 90 p.
35. **Tatjana Nahtman.** Permutation invariance and reparameterizations in linear models. Tartu 2004. 91 p.
36. **Märt Möls.** Linear mixed models with equivalent predictors. Tartu 2004. 70 p.
37. **Kristiina Hakk.** Approximation methods for weakly singular integral equations with discontinuous coefficients. Tartu 2004, 137 p.
38. **Meelis Käärik.** Fitting sets to probability distributions. Tartu 2005, 90 p.
39. **Inga Parts.** Piecewise polynomial collocation methods for solving weakly singular integro-differential equations. Tartu 2005, 140 p.
40. **Natalia Saealle.** Convergence and summability with speed of functional series. Tartu 2005, 91 p.
41. **Tanel Kaart.** The reliability of linear mixed models in genetic studies. Tartu 2006, 124 p.
42. **Kadre Torn.** Shear and bending response of inelastic structures to dynamic load. Tartu 2006, 142 p.
43. **Kristel Mikkor.** Uniform factorisation for compact subsets of Banach spaces of operators. Tartu 2006, 72 p.
44. **Darja Saveljeva.** Quadratic and cubic spline collocation for Volterra integral equations. Tartu 2006, 117 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
46. **Annely Mürk.** Optimization of inelastic plates with cracks. Tartu 2006. 137 p.
47. **Annemai Raidjõe.** Sequence spaces defined by modulus functions and superposition operators. Tartu 2006, 97 p.
48. **Olga Panova.** Real Gelfand-Mazur algebras. Tartu 2006, 82 p.

49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
50. **Margus Pihlak.** Approximation of multivariate distribution functions. Tartu 2007, 82 p.
51. **Ene Käärik.** Handling dropouts in repeated measurements using copulas. Tartu 2007, 99 p.
52. **Artur Sepp.** Affine models in mathematical finance: an analytical approach. Tartu 2007, 147 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
54. **Kaja Sõstra.** Restriction estimator for domains. Tartu 2007, 104 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
57. **Evely Leetma.** Solution of smoothing problems with obstacles. Tartu 2009, 81 p.
58. **Ants Kaasik.** Estimating ruin probabilities in the Cramér-Lundberg model with heavy-tailed claims. Tartu 2009, 139 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
60. **Indrek Zolk.** The commuting bounded approximation property of Banach spaces. Tartu 2010, 107 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
63. **Marek Kolk.** Piecewise Polynomial Collocation for Volterra Integral Equations with Singularities. Tartu 2010, 134 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
65. **Larissa Roots.** Free vibrations of stepped cylindrical shells containing cracks. Tartu 2010, 94 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
68. **Olga Liivapuu.** Graded q-differential algebras and algebraic models in noncommutative geometry. Tartu 2011, 112 p.
69. **Aleksei Lissitsin.** Convex approximation properties of Banach spaces. Tartu 2011, 107 p.
70. **Lauri Tart.** Morita equivalence of partially ordered semigroups. Tartu 2011, 101 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.

72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.
74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
75. **Nadežda Bazunova.** Differential calculus $d^3 = 0$ on binary and ternary associative algebras. Tartu 2011, 99 p.
76. **Natalja Lepik.** Estimation of domains under restrictions built upon generalized regression and synthetic estimators. Tartu 2011, 133 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
80. **Marje Johanson.** $M(r, s)$-ideals of compact operators. Tartu 2012, 103 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
82. **Vitali Retšnoi.** Vector fields and Lie group representations. Tartu 2012, 108 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
85. **Erge Ideon**. Rational spline collocation for boundary value problems. Tartu, 2013, 111 p.
86. **Esta Kägo.** Natural vibrations of elastic stepped plates with cracks. Tartu, 2013, 114 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
88. **Boriss Vlassov.** Optimization of stepped plates in the case of smooth yield surfaces. Tartu, 2013, 104 p.
89. **Elina Safiulina.** Parallel and semiparallel space-like submanifolds of low dimension in pseudo-Euclidean space. Tartu, 2013, 85 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
93. **Kerli Orav-Puurand.** Central Part Interpolation Schemes for Weakly Singular Integral Equations. Tartu, 2014, 109 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multiparty computation. Tartu, 2015, 201 p.

95. **Kaido Lätt.** Singular fractional differential equations and cordial Volterra integral operators. Tartu, 2015, 93 p.
96. **Oleg Košik.** Categorical equivalence in algebra. Tartu, 2015, 84 p.
97. **Kati Ain.** Compactness and null sequences defined by ℓ_p spaces. Tartu, 2015, 90 p.
98. **Helle Hallik.** Rational spline histopolation. Tartu, 2015, 100 p.
99. **Johann Langemets.** Geometrical structure in diameter 2 Banach spaces. Tartu, 2015, 132 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
105. **Md Raknuzzaman.** Noncommutative Galois Extension Approach to Ternary Grassmann Algebra and Graded q-Differential Algebra. Tartu, 2016, 110 p.
106. **Alexander Liyvapuu.** Natural vibrations of elastic stepped arches with cracks. Tartu, 2016, 110 p.
107. **Julia Polikarpus.** Elastic plastic analysis and optimization of axisymmetric plates. Tartu, 2016, 114 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.
113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.