U N I V E R S I T Y OF T A R T U
Faculty of Mathematics and Computer Science
Institute of Computer Science

Dan Bogdanov

How to securely perform computations on secret-shared data

Master's Thesis

Supervisor: Jan Willemson, PhD
Instructor: Sven Laur, MSc

TARTU 2007

Contents

1 Introduction

1.1 Problem statement

Databases containing personal, medical or financial information about an individual are usually classified as sensitive. Often the identity of the person is somehow stored in the database, whether by name, personal code or a combination of attributes. In many countries it is illegal to process such data without a special license from the authorities. Such protection is needed to preserve the privacy of individuals and prevent abuse of the data.

This level of protection is a problem for research organisations, that cannot learn global properties or trends from collected data. It also prevents government organisations from providing accurate demographics reports and managing medical registries about the population. Data analysis restriction is not the only problem for these organisations but a solution is nevertheless expected.

In this thesis we address a simplified version of the problem. Assume that we have asked p people q sensitive questions. By collecting the answers we obtain a matrix $\mathcal{D}$ with p rows and q columns denoted that represents our data. Our goal is to devise a method for computing aggregate statistics from this matrix without compromising the privacy of a single person, that is, revealing values in matrix $\mathcal{D}$.

1.2 General solutions

To give an impression of the problem we discuss three possible solutions and evaluate them with regard to security and usability for data analysis. Let us consider a group of parties that gather data and construct the matrix. Let M be the data miner that is interested in global patterns. In the common scenario, all parties give their data to M that constructs $\mathcal{D}$ and runs aggregation and data mining algorithms on it. The parties have to trust that M will not use the data for selfish gains. They have no way of ensuring the privacy of people who have provided the data, because the miner requires control over the complete database to perform computations.

We want to keep a specific party from having the complete database. This way the parties will not have to trust a single entity. We now present three generic approaches for solving this problem. It is important to note, that from the three presented solutions only the last one offers an acceptable balance between privacy and data utility.

Solution 1: Distribution of rows. We can divide the $p \times q$ data matrix $\mathcal{D}$ into smaller matrices $\mathcal{D}_1, \ldots, \mathcal{D}_t$ so that each smaller matrix contains some rows from $\mathcal{D}$. We can then distribute matrices $\mathcal{D}_1, \ldots, \mathcal{D}_t$ to independent miners $M_1, \ldots, M_t$

that compute results based on their part of the data and forward them to the "master analyst" M that combines the results of the miners. Figure 1 illustrates this analysis process.

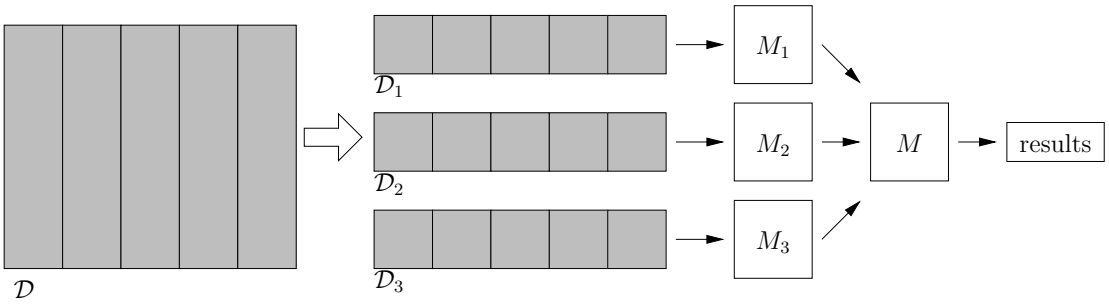

Figure 1: Database distribution by rows between three miners

Unfortunately, this solution does not provide privacy of the data, as each party will have one or more complete rows of the matrix $\mathcal{D}$. One row of the matrix contains data about a single person who answered the questions so the party with this row will have access to all the data about this person. If only a subset of the miners are malicious, then the risk of data compromise is smaller than it would be if a single miner processed the data. Still, this solution is definitely not secure for use in real life.

Solution 2: Distribution of columns. A similar idea is to divide the matrix $\mathcal{D}$ so that matrices $\mathcal{D}_1, \ldots, \mathcal{D}_t$ contain columns from the original matrix $\mathcal{D}$. This allows us to keep the data identifying the person in a separate database from the sensitive data. Figure 2 illustrates such a scenario.

This solution decreases the usability of the data, because one miner has access only to some attributes. For example, this could keep us from finding reliable aggregations or association rules based on all relevant attributes, because some of them might not be available for a given miner. Since data analysis is still the main goal, we have to find some other way to provide security.

Distributing the data to columns also does not provide privacy, because some combinations of attributes may uniquely identify a person. For example, in a small village there may be just one male person with a high salary.

Solution 3: Distribution of values. Our solution is based on the third approach. Instead of distributing matrix $\mathcal{D}$ into smaller matrices we distribute its values between miners $M_1, \ldots, M_t$ so that no single miner nor a pair of them can find the original value given their parts of this value. We use *secret sharing* to distribute values in the original matrix into t parts. These parts will form matrices $D_1, \ldots, D_t$ as shown on Figure 3.

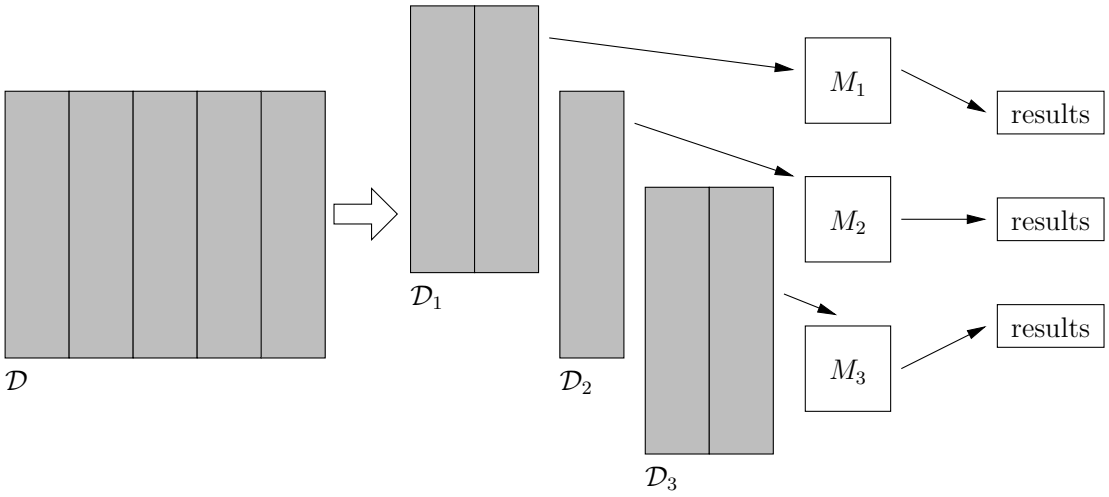

Figure 2: The database is distributed by columns between three miners

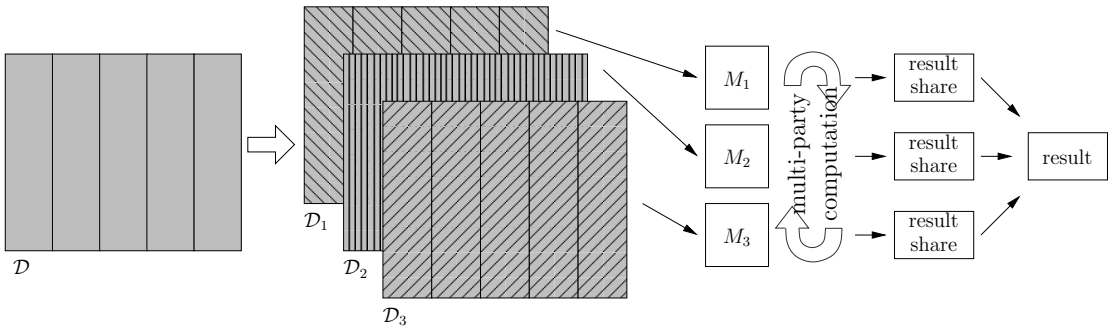

Figure 3: The values in the database are distributed between three miners

The miners will then run *secure multiparty computation* protocols to perform computations on the data without reconstructing the original values. When aggregation results are computed, each miner sends its part of the results to the client, that combines the parts to form the complete results. The miners must check their inputs and allow only justified computations and reject queries that would compromise privacy.

In this work we present a theoretical framework to complete this task. We also demonstrate the feasibility of the proposed solution by building an implementation of the framework.

1.3 Outline of this thesis

This thesis describes the concepts used in designing our solution and the derived implementation. The work is structured into chapters as described below.

- Chapter 2 gives an overview of secure multiparty computation. The chapter contains basic definitions and an overview of results in this area. We also describe a method of proving security of protocols—the ideal versus real world model.

- Chapter 3 describes secret sharing schemes. In particular, we give an analysis of Shamir's secret sharing scheme. We show how the homomorphic property of a secret sharing scheme allows us to perform computations with shared data. In the conclusion of the chapter we show the similarity between coding theory and secret sharing and identify coding schemes suitable for building secret sharing schemes.

- Chapter 4 presents the theoretical framework of our solution. We list our design goals along with engineering choices made according to those goals. We also give an overview of the security model associated with our system and present protocols that perform basic operations in our framework.

- Chapter 5 is dedicated to our implementation—Sharemind secure computing platform. We describe the capabilities of the platform and the computation environment it provides.

- Chapter 6 presents practical results achieved on our implementation and discusses the feasibility of the approach.

1.4 Author's statement

In this section we list the author's original contributions to this thesis. Chapters 2 and 3 give an overview of known results in multiparty computation and secret sharing related to this work. The author developed and verified the framework and protocols described in Chapter 4 in co-operation with Sven Laur. The main contribution of the author is creating an implementation of the framework—the SHAREMIND platform that is described in Chapter 5. The author is responsible for the architecture and design of the software and also the symmetrical implementation of protocols—during computation each party runs the same code with minor exceptions. In Chapter 6 the author presents the experiment results achieved with the SHAREMIND platform. The source code of SHAREMIND is released under version 2 of the GNU General Public License.

2 Secure multiparty computation

2.1 Definitions

Let us consider an interactive computing scenario with more than one party. Each party has some input value x_i and together they want to compute some output based on the inputs of all parties. To achieve that, the parties must use multiparty computation which means exchanging messages and performing local computations until all parties have learned the desired output.

Secure multiparty computation is the evaluation of a function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ in such a way that the output is correct and the inputs of the parties are kept private. Each party P_i will get the value of y_i and nothing else. For further discussion on multiparty computation see the manuscript [CD04]

A classical example of multiparty computation is the millionaire problem. Assume that Alice and Bob are millionaires who would like to know, which one of them is richer without revealing their exact wealth.

More precisely, let Alice's wealth be x_1 and Bob's wealth be x_2. Then the function we need to evaluate is "greater than", that is, we need to find out, if $x_1 > x_2$ without anyone else knowing x_1 or x_2. There are various solutions to this problem. A sketch for a solution was presented together with the problem introduction by Yao [Yao82].

To allow parties to perform computations together, we must provide them with communication channels. There are two main communication models used in secure multiparty computation: *information-theoretic model* and *cryptographic model*. In the information-theoretic model all nodes have private communication channels between them. In the cryptographic model the adversary is provided with read-only access to all the messages from the traffic between honest nodes. The adversary may not modify the messages exchanged between honest nodes.

The cryptographic model is secure if the adversary cannot break the cryptographic primitive and read the messages. The information-theoretic model is stronger, as even a computationally unbounded adversary cannot read the messages exchanged between honest nodes.

Note, that our definitions of security are presented in the information-theoretic model, but our real-world framework operates in the cryptographic model. We achieve reasonable security in the real world by using AES encryption and message authentication codes to protect the traffic between honest nodes and even allow the adversary to add new messages to the traffic. The encryption will prevent the adversary from reading the messages and authentication codes will help the parties distinguish malicious messages from legal ones.

Communication between parties can be *synchronous* or *asynchronous*. In the synchronous mode nodes have synchronised clocks which allows us to design pro-

tocols with rounds. Messages sent each round will be delivered before the next round begins. The asynchronous model is more complex as it has no guarantees on message delivery time. If we do not have guarantees for message delivery, we cannot demand that the protocol reaches a certain step at all.

A standard way for modelling the adversary in secure multiparty computation is by the use of malicious parties. The adversary may "corrupt" any number of nodes before the protocol starts. In the beginning of the protocol the honest players do not know, which nodes are corrupted and which are not.

In the case of *passive corruption* the adversary can read all the data held, sent and received by the node. If the corruption is *active*, the adversary has complete control over the node. If the adversary is *static*, then the set of corrupt nodes remains the same for the whole duration of the protocol. The adversary may also be *adaptive* and corrupt new nodes during the execution of the protocol.

We must limit the adversary to keep secure protocols possible. If all parties are corrupt, we have no hope to complete the computation of the function. Therefore, we restrict the adversary to corrupting only a proper subset of all the parties in $P = \{P_1, \ldots, P_n\}$. To model this behaviour we define the *access structure* and the *adversary structure*.

Given a set of parties P we call $\Gamma_P \subset \mathcal{P}(P)$ an *access structure* on P. An access structure is assumed to be *monotone* if and only if $\emptyset \notin \Gamma_P$ and $\forall B$ $[B \in \Gamma_P, B \subseteq B' \subseteq P \Rightarrow B' \in \Gamma_P]$. We call $\mathcal{A} \subset \mathcal{P}(P)$ an *adversary structure*, if $\mathcal{P}(P) \backslash \mathcal{A}$ is a monotone access structure. Intuitively, the adversary structure $\mathcal{A}$ contains all possible sets of parties C that can be corrupted by the adversary. If the adversary can corrupt a set of parties C it can also corrupt parties in all subsets of C. Also, for each set of parties B in the access structure all supersets of B also belong to the access structure.

2.2 Security goals and feasibility results

Our goal in this work is to build a tool for performing secure computations. The most important security aspect for us is privacy. We want to keep the data private from as many parties as possible. To be precise, we want to perform data analysis without showing actual values to the analyst. Instead, we compute the requested aggregations and provide only the results. Another aim is to build a fast implementation of the tool, so we value efficiency in schemes and protocols. In this section we discuss the various options for performing secure multiparty computation.

We only consider approaches with more than one party, because providing privacy in a scenario with a single computing party is impossible. If the adversary takes over the party, we have lost everything. Using more than one computing party in secure data processing is a standard approach. This is normally achieved

by allowing the computing nodes to communicate with each other by using, for example, a computer network.

The term *multiparty computation*, in fact, applies to cases where three or more parties do the work. Two-party computation is less practical because it is less efficient than computation with three or more parties. Usually secure and efficient two-party computation schemes rely on homomorphic encryption, oblivious transfer or some other computationally expensive primitive. Oblivious transfer is usually built on trapdoor permutations and one-way functions [Gol04]. The same building blocks are used in public key encryption schemes that are several orders of magnitude slower than symmetric cryptography that can be used to build multiparty computation systems.

To achieve a reasonable balance between security and efficiency, we work in the *semi-honest model* that is also known as the *honest-but-curious* model. In this model we assume that the parties follow the protocol, but they also try to compute secret values based on the data available to them. This model suits well with our main goal of protecting the privacy of the data.

Literature contains possibility results for multi-party computation in various security models. In their well-known works Ben-Or, Goldwasser, Wigderson [BOGW88] and Chaum, Crépeau, Damgård [CCD88] showed the existence of information theoretically secure general multiparty computation protocols. Both papers proved that there exists a correct, private and robust polynomial time protocol evaluating a function f if and only if the adversary corrupts at most $k < \frac{n}{3}$ players. However, in the semi-honest model the number of corrupt parties k must be less than $\frac{n}{2}$. They also showed that these bounds are optimal and we cannot do better. Both papers use Shamir's secret sharing scheme to build their protocols. The differences are in protocol construction, where Ben-Or *et al* rely on properties of error correcting codes and Chaum *et al* use distributed commitments and zero-knowledge. Ben-Or *et al* achieve perfect correctness while Chaum *et al* have a small margin for error.

These results set the stage for our framework—we see how much security we can achieve with a set of parties. Since every added party increases traffic, we want to keep their number down. Moreover, every additional party also increases the costs of protecting the system from an external attacker. Based on these arguments, we consider a three-party secure computation scenario optimal for our solution. In this model, we can guarantee security if more than half of the parties are honest. In our case this means that we can allow one honest-but-curious party in our system.

2.3 The ideal versus real world paradigm

A common method in defining protocol security in multiparty computation is the ideal versus real world model [Gol04]. We consider two separate worlds—the real world where the protocol is implemented, executed, and attacked, and the ideal world that contains the specification of the protocol's behaviour. When the protocol is properly described and set up in both worlds we can say that a protocol is secure, if its output in the real world cannot be distinguished from its output in the ideal world.

To facilitate this we need a formal framework for describing the protocol in our ideal world. To start, the ideal world requires some incorruptible party to have any chance of being secure at all. In this model it is the trusted third party F that implements the *ideal functionality*. F always acts according to the protocol.

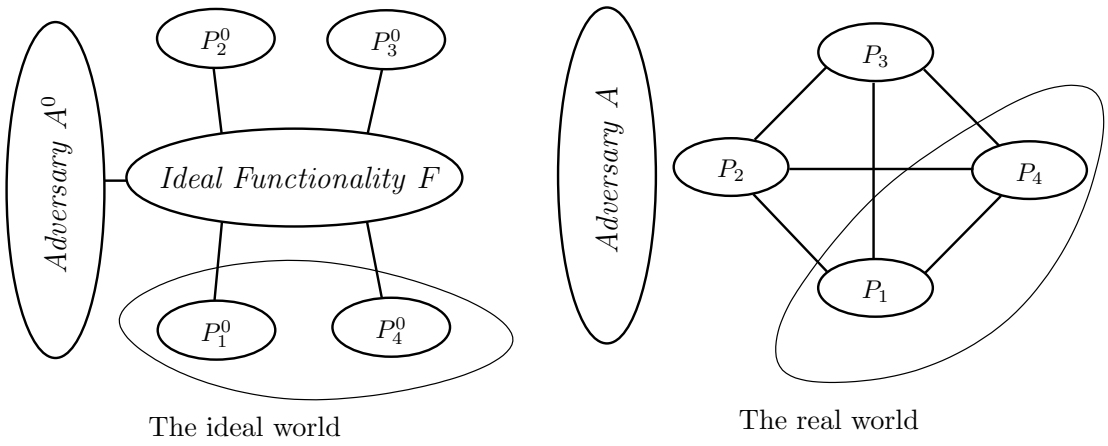

Figure 4: Example of a ideal vs real world scenario

Figure 4 shows a structural comparison of the ideal and real world. In this example parties P_1 and P_4 are corrupted by the adversary. This means that, in both worlds the adversary can read the input and control the output of P_1 and P_4, but with noticeable differences. In the ideal world all communication passes through F whereas in the real world all communication is performed directly between the nodes. By default, we assume that channels cannot be tapped and messages cannot be altered or delayed in either world.

We assume that corrupt parties also follow the protocol and send their messages in a reasonable time. Otherwise a corrupt party might just abort the protocol and prevent everyone from computing the results. This is consistent with the other properties of the described semi-honest model.

The interface of F consists of input and output ports for each other party and one input and output port for the adversary. Figure 5 shows an example of F with ports for four parties and the adversary.

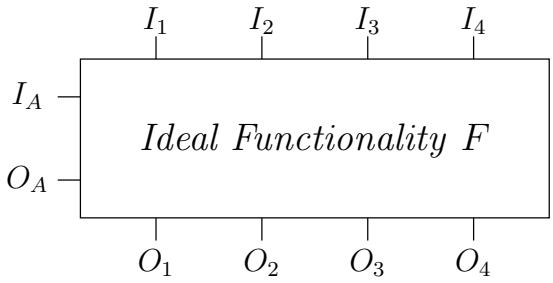

Figure 5: Example of the interface of the trusted third party F

Each round in the ideal world has the following structure:

1. For each uncorrupted party P_k, the trusted third party F reads the corresponding input from port I_k.
2. For each corrupt party P'_k, the trusted third party F reads its input from I_A—the input port of the adversary.
3. F processes the inputs and computes the outputs.
4. F outputs the results for each uncorrupted party P_k on port O_k.
5. F outputs the results for each corrupted party P'_k on port O_A.

Rounds in the real world have the following structure:

1. All parties send messages to other parties according to protocol.
2. All parties receive messages.
3. Corrupted parties send all internal data including coinflips, computation results and received messages to the adversary A.

Examples of protocol structure in both worlds are shown on Figure 6 and Figure 7. In this setting there are two parties, P_1 and P_2. P_1 has been corrupted by the adversary. The adversary has total control over the communication of this party. A protocol between nodes P_1 and P_2 is executed both in the real world and in the ideal world. Both parties have an input value x_i and output value y_i. The adversary has an output value y_A. The trusted party has no inputs or outputs.

The protocol runs in n rounds in the real world. In the ideal world the number of rounds m can be smaller than n and it is often preferred to keep security proofs simple. Usually, the ideal world protocol has only one round. To compute $f(x_1, \ldots, x_n)$ the parties send their inputs to the trusted third party and obtain

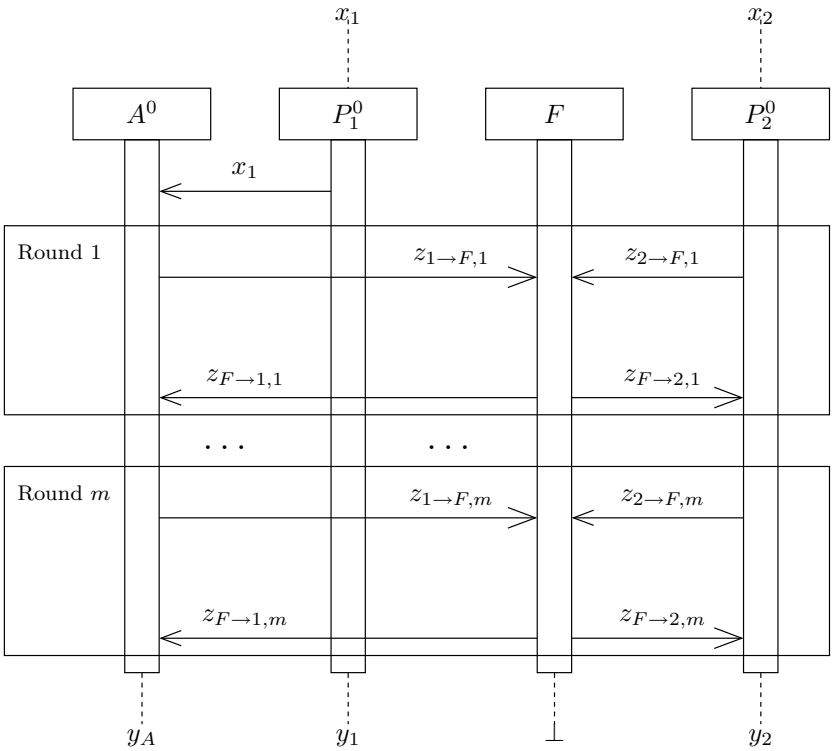

Figure 6: Structure of a protocol in the ideal world

$f(x_1, \ldots, x_n)$ in reply. For an example of a protocol with one round in the ideal world and three rounds in the real world, see Section 2.4.

Before the rounds start, the adversary learns the input of P_1 because it knows everything P_1 knows. In each round the parties send one message and receive one message. The messages are denoted $z_{i\to j,r}$ where i is the index of the sender and j is the index of the recipient and r is the current round. An exception to this rule is the trusted third party, that has no index—the letter F is used instead.

Figure 6 shows how the protocol runs in the ideal world. Each round, the trusted third party F reads all inputs, processes them and outputs the results. Note that in the ideal model the adversary sends and receives messages on behalf of P_1. This is because P_1 is corrupted and its traffic is routed through ports I_A and O_A of F.

Now we will examine what goes on in the real world, see Figure 7. In the real world P_1 does its own messaging, but reports everything it learns to the adversary. Since there is no trusted party to perform computations, the protocol must also specify the computation of F. In either world the honest parties have no idea that when they are communicating with P_1 they are actually talking to the adversary. F is essentially a trusted party that handles communication in the ideal world.

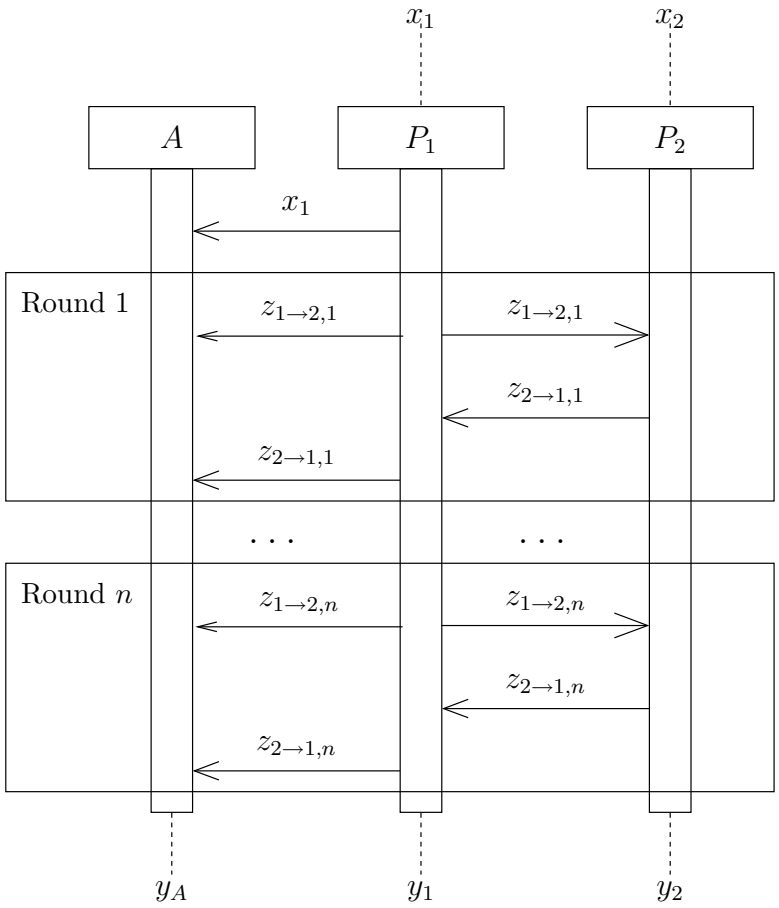

Figure 7: Structure of a protocol in the real world

From the execution of the rounds it can be seen, how the adversary handles the communication for all corrupted parties through the special ports of F.

We notice a property of the ideal versus real world paradigm—if the protocol is correctly defined, we can use the attacks formed in the ideal world also in the real world.

Finally, we give the definition of security for our protocols. This definition forms the core for our security model as the security proofs for all protocols in our framework are based on it.

Definition 2.1. *A protocol π is* perfectly secure*, if for any adversary A in the real world there exists A^0 in the ideal world so that for any inputs $x_1, \ldots, x_n$ the output distributions of $A, P_1, \ldots, P_n$ and $A^0, P_1^0, \ldots, P_n^0$ coincide.*

2.4 How to simulate the real world

The outputs of the parties and the adversary must have the same distribution in both the ideal and the real world. One way of achieving this is to use a *simulator wrapper* S which translates the traffic of A to the ideal world setting, so that A cannot distinguish its surroundings from the real world. If we just place A into the ideal world, then it expects to receive protocol messages which are not provided in the ideal world because the computation is performed by the trusted third party F and each party receives only the results. The adversary A must be able to communicate with S in exactly the same way as in the real world. From the other side of the simulator the trusted party looks at S and A and sees the ideal adversary A^0. Figure 8 illustrates how the new ideal adversary A^0 is constructed.

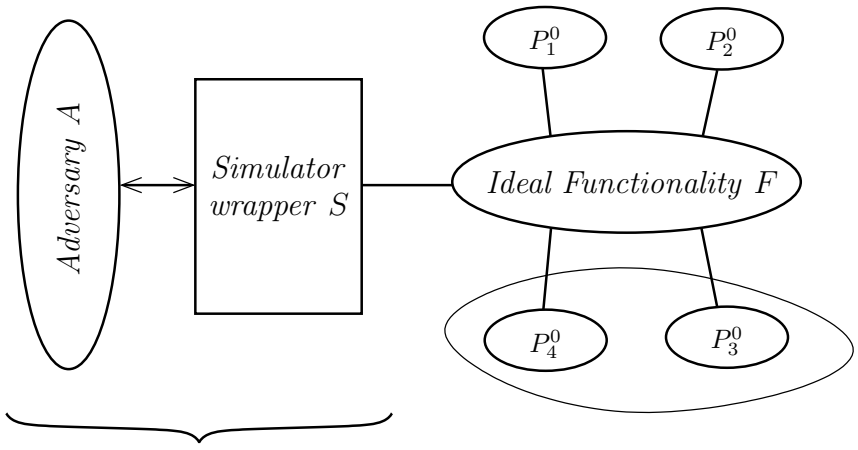

Figure 8: Construction of the ideal adversary A^0

We will now look at some examples of constructing an ideal adversary for an actual real world protocol. First we set the stage. Assume that we have parties P_1, P_2 and P_3 and they want to compute a function $f(x_1, x_2, x_3) = (y_1, y_2, y_3)$. One of the nodes—$P_1$—is corrupted by the adversary that wants to learn the outputs of other parties.

The first example we consider somewhat corresponds to a secret sharing scenario. In the real world each node receives k messages with a uniform distribution. The nodes give no output. Simulating this protocol is easy—the simulator generates k values with a uniform distribution and passes them to A. Since all messages are from a uniform distribution, A has no way for distinguishing the ideal world from the real one.

In the second generic example we have a computation protocol and we want to make it perfectly secure. For this we need to show that the protocol corresponds

to Definition 2.1, that is, for each real world adversary A there exists an ideal adversary A^0 so that $\Pr[(y_{A^0}, y_1, y_2, y_3)|\mathsf{IDEAL}] = \Pr[(y_A, y_1, y_2, y_3)|\mathsf{REAL}]$. One way to achieve this is to compose the protocols so, that all messages received by parties are independent and from a uniform distribution. This property makes the construction of the simulator wrapper considerably easier as we demonstrate in the following example.

Assume that we want to compute $f(x_1, x_2, x_3) = x_1 + x_2 + x_3 = y$. All values are from the residue class $\mathbb{Z}_N$ and all operations are performed mod N. Each party has one uniformly distributed input value x_i. The following protocol is an instantiation of the Benaloh addition protocol [Ben86]. In the real world, each node P_i performs the following steps:

1. Uniformly generate three values $x_{i1}, x_{i2}, x_{i3} \leftarrow \mathbb{Z}_N$ so that $x_{i1} + x_{i2} + x_{i3} = x_i$. A simple way to do this is to generate two values $x_{i1}, x_{i2} \leftarrow \mathbb{Z}_N$ and compute $x_{i3} = x_i - x_{i1} - x_{i2}$. Note, that we can also do this symmetrically by generating any other two values of the three and computing the third value by subtraction.

2. Send x_{ij} to party P_j

3. Compute the sum $c_i = x_{1i} + x_{2i} + x_{3i}$

4. Publish c_i by sending it to other parties

5. Compute $y = c_1 + c_2 + c_3$

In the following proofs we choose one party P_1 and prove the result for messages received by it. Since the protocol is symmetrical, the proofs for other parties are similar. Note, that in the following proof we explicitly show the uniformity and independence of protocol messages and how these properties are used to construct the simulator.

Lemma 2.1. *The presented addition protocol with three parties is correct.*

Proof. To prove correctness we need to show that $x_1 + x_2 + x_3 = c_1 + c_2 + c_3$. For that we expand the right hand side of the equation as follows:

$$\begin{aligned} c_1 + c_2 + c_3 &= (x_{11} + x_{21} + x_{31}) + (x_{12} + x_{22} + x_{32}) + (x_{13} + x_{23} + x_{33}) \\ &= (x_{11} + x_{12} + x_{13}) + (x_{21} + x_{22} + x_{23}) + (x_{31} + x_{32} + x_{33}) \\ &= x_1 + x_2 + x_3 \ . \end{aligned}$$

□

Lemma 2.2. *The protocol messages x_{21} and x_{31} received by P_1 in step 3 are uniformly distributed in $\mathbb{Z}_N \times \mathbb{Z}_N$.*

Proof. We know that x_{i1} and x_{i2} are generated from a uniform distribution and x_{i3} is computed by modifying x_i by the sum of x_{i1} and x_{i2} that is also a uniformly distributed value. It follows that two of these values are always from a uniform distribution and independent. □

Lemma 2.3. *The protocol messages c_2 and c_3 received by P_1 in step 5 are uniformly chosen from the set $\mathcal{C}(y, c_1) = \{(c_2, c_3) : c_1 + c_2 + c_3 = y\}$ for any fixed pair of step 3 messages.*

Proof. The party P_1 receives sums $c_2 = x_{12} + x_{22} + x_{32}$ and $c_3 = x_{13} + x_{23} + x_{33}$. Although P_1 knows the values x_{12} and x_{13} these sums are uniformly distributed, because their addends x_{22} and x_{33} are uniformly distributed in $\mathbb{Z}_N \times \mathbb{Z}_N$ for any fixed pair of messages x_{21} and x_{31} received in the third step. According to Lemma 2.1 the protocol correctly computes the sum y and therefore $(c_2, c_3) \in \mathcal{C}(y, c_1)$. □

Theorem 2.1. *The given addition protocol is correct and perfectly secure in the semi-honest model.*

Proof. The correctness was proven in Lemma 2.1. We now prove that the protocol is perfectly secure and for that we will again assume that the adversary has corrupted the node P_1. The simulator S knows x_1 which is the the input of P_1, and the computed sum y which is provided by the trusted third party F.

The simulator S starts by simulating the behaviour of P_1 and computing values x_{11}, x_{12} and x_{13}. In the first round the adversary expects to see values x_{21} and x_{31} from parties P_2 and P_3. The simulator S uniformly generates two values $\hat{x}_{21}, \hat{x}_{31} \leftarrow \mathbb{Z}_N \times \mathbb{Z}_N$. The adversary A cannot distinguish the situation from the real world, because in Lemma 2.2 we showed that all values x_{ij}, $(i \neq j)$ received by P_1 in the first round are from a uniform distribution.

In the second round the adversary expects to see values c_2 and c_3 from other parties. The simulator computes $c_1 = x_{11} + x_{21} + x_{31}$ and proceeds by uniformly generating $\hat{c}_2$ and computing $\hat{c}_3 = y - c_1 - \hat{c}_2$. This way we have uniformly generated two values from $\mathcal{C}(y, c_1)$. Again, the adversary cannot distinguish between values it receives in the ideal world from those it receives in the real world, because also according to Lemma 2.3 these values are from a uniform distribution and give the expected sum.

The simulator S can now run the adversary A and pass the computed values to it and because A cannot distinguish the messages given by S from the messages it receives in the real world, its output cannot be distinguished from what it gives in the real world. We have constructed a suitable simulator S and thus proven the theorem. The cases for parties P_2 and P_3 are similar because of the symmetry of the protocol. □

Corollary 2.1. *If at any moment in our protocol we have securely computed* $c_1 + c_2 + c_3 = f(x_1, x_2, x_3)$ *so that* $c_1, \ldots, c_3$ *are chosen uniformly from the set* $\mathcal{C}(y) = \{(c_1, c_2, c_3) : c_1 + c_2 + c_3 = y\}$*, we can publish* $c_1, \ldots, c_3$ *to compute* f*.*

Proof. The scenario describes exactly the situation we encountered in the proof of Theorem 2.1. To build the simulator for party P_1 we uniformly generate $c_2 \leftarrow \mathbb{Z}_N$ and then compute $c_3 = y - c_1 - c_2$. The other cases are symmetrical. □

This result is an important building block for protocol construction. We have reduced computing any f to computing additive shares of it.

3 Homomorphic secret sharing

3.1 Concept of secret sharing

Secret sharing is a technique for protecting sensitive data, such as cryptographic keys. It is used to distribute a secret value to a number of parts—*shares*—that have to be combined together to access the original value. These shares can then be given to separate parties that protect them using standard means, e.g., memorize, store in a computer or in a safe. Secret sharing is is used in modern cryptography to lower the risks associated with compromised data.

Sharing a secret spreads the risk of compromising the value across several parties. Standard security assumptions of secret sharing schemes state that if an adversary gains access to any number of shares lower than some defined threshold, it gains no information of the secret value. The first secret sharing schemes were proposed by Shamir [Sha79] and Blakley [Bla79].

Our interest in secret sharing is inspired by its usefulness in secure multiparty computation. Secret sharing directly helps us in preserving the privacy of our data. In this chapter we explore secret sharing to determine its versatility for us in a computing environment. We are mostly interested in the possibility of performing operations with shares so we do not have to reconstruct original values all the time.

Definition 3.1. *Let the secret data be a value s. An algorithm $\mathbf{S}$ defines a k-out-of-n* threshold secret sharing scheme*, if it computes $\mathbf{S}(s) = [s_1, \ldots, s_n]$ and the following conditions hold [Sha79, Dam02]:*

1. ***Correctness:*** *s is uniquely determined by any k shares from $\{s_1, \ldots, s_n\}$ and there exists an algorithm $\mathbf{S}'$ that efficiently computes s from these k shares.*

2. ***Privacy:*** *having access to any $k-1$ shares from $\{s_1, \ldots, s_n\}$ gives no information about the value of s, i.e., the probability distribution of $k-1$ shares is independent of s.*

A secret sharing scheme is *homomorphic* if it is possible to compute new valid shares from existing shares.

Definition 3.2. *Let s and t be two values and $[\![s]\!] = [s_1, \ldots, s_n]$ and $[\![t]\!] = [t_1, \ldots, t_n]$ be their shares. A secret sharing scheme is* $(\oplus, \otimes)$-homomorphic *if shares $[(s_1 \otimes t), \ldots, (s_n \otimes t)]$ uniquely determine the value $s \oplus t$.*

If individual shares are from a uniform distribution it can be shown that secret sharing is secure in a multiparty computation setting. Indeed, the protocol is

simple—one party sends values from a uniform distribution to other parties in the system. In the ideal world this means the trusted third party F outputs nothing. The simulator is easy to build—it just generates a value from a uniform distribution and passes it to the adversary. Again, the values are from the same distribution and the adversary cannot distinguish between them.

To illustrate the concept of secret sharing, we use the following classical example [Sha79]. Assume that there is a corporation where the management needs to digitally sign cheques. The president can sign cheques on his or her own, the vice presidents need at least another member of the board to give a signature and other board members need at least two other managers to sign a cheque.

We can solve this problem by sharing the secret key needed for giving a signature with a 3-out-of-n threshold secret sharing scheme, where n is the required number of shares. We give the company president three shares, so he or she can sign cheques alone. Vice presidents get two shares each, so that they need the agreement of another manager to give a signature. Other members of the board get one share per member, so that three of them are needed for a signature.

The signing device is completely secure as it does not contain any secret information. It requires the members of the board to provide three shares to retrieve the signature key. This key is used for a single signature and then forgotten so the next signature will again require three shares of the key. If a malicious adversary coerces one manager to sign a cheque, then it has to be the president of the corporation. Otherwise the adversary will have to persuade more than one member of the board.

This example naturally leads us to the notion of threshold signature schemes that allow us to compute a signature without reconstructing the key itself. The notion was introduced by Desmedt [Des88] and various signature schemes have been proposed that do not require the presence of a key but only parts of it. Examples include threshold variants of ElGamal and RSA signature schemes presented by Desmedt and Frankel [DF89] and Shoup [Sho00].

3.2 Mathematical foundations of secret sharing

3.2.1 Polynomial evaluations

We start by describing some basic properties of polynomials. Let us consider a ring $\mathbf{R}$ and denote the set of all polynomials over $\mathbf{R}$ by $\mathbf{R}[x]$. Let $f(x) = f_0 + f_1 x + \cdots + f_k x^k$ be a polynomial in $\mathbf{R}[x]$. We also fix a vector $a = [a_0, \ldots, a_n] \in \mathbf{R}^n$ so that all values a_i are different and nonzero. Then we define the polynomial evaluation mapping $\mathbf{eval} : \mathbf{R}[x] \to \mathbf{R}^n$. as follows. We evaluate the polynomial $f(x)$ on the vector a and present the result as a vector.

$$\mathbf{eval}(f) := [f(a_0), \ldots, f(a_n)]^T \quad .$$

In the following theorem operations between vectors in $\mathbf{R}^n$ are defined elementwise. That is, if $u, v \in \mathbf{R}^n$ and $\oplus$ is a binary operator, then:

$$u \oplus v := [(u_1 \oplus v_1), \ldots, (u_n \oplus v_n)]^T \quad .$$

Theorem 3.1. *For any two polynomials f and g in $\mathbf{R}[x]$ and a scalar value $r \in \mathbf{R}$ the following conditions hold:*

(i) Additivity: $\mathbf{eval}(f + g) = \mathbf{eval}(f) + \mathbf{eval}(g)$.

(ii) Multiplicativity: $\mathbf{eval}(f \cdot g) = \mathbf{eval}(f) \cdot \mathbf{eval}(g)$.

(iii) Multiplicativity w.r.t. scalar values: $\mathbf{eval}(r \cdot f) = r \cdot \mathbf{eval}(f)$

The mapping **eval** *is a linear transformation.*

Proof. The conditions hold because of the duality with the respective polynomial operations:

(i) Additivity: $(f + g)(a) = f(a) + g(a)$

(ii) Multiplicativity: $(f \cdot g)(a) = f(a) \cdot g(a)$

(iii) Multiplicativity w.r.t. scalar values: $(r \cdot f)(a) = r \cdot f(a)$

The conclusion that the mapping is a linear transformation directly follows from the above conditions. Thus we have shown that **eval** is a linear mapping between the evaluation positions of the polynomial and the result vector. □

We will now give a further analysis of the properties of this mapping. Let $\vec{f} = [f_0, \ldots, f_k]$ be the array of coefficients of the polynomial f. Note that in further discussion we consider a polynomial f being equivalent to the vector of its coefficients.

We now compute the vector $\vec{y} = \mathbf{eval}(f) = [f(a_0), \ldots, f(a_n)]^T$.

$$\begin{aligned}\vec{y} &= \sum_{i=0}^{k} f_i \mathbf{eval}(x^i) = \sum_{i=0}^{k} f_i \left[a_0^i, \ldots, a_n^i\right]^T \\ &= f_0 \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} + f_1 \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} + \cdots + f_k \begin{bmatrix} a_0^k \\ a_1^k \\ \vdots \\ a_n^k \end{bmatrix} = \begin{bmatrix} f_0 + f_1 a_0 + \cdots + f_k a_0^k \\ f_0 + f_1 a_1 + \cdots + f_k a_1^k \\ \cdots \\ f_0 + f_1 a_n + \cdots + f_k a_n^k \end{bmatrix}.\end{aligned}$$

We notice that the vector $\vec{y}$ is actually the product of a matrix and another vector.

$$\begin{bmatrix} f_0 + f_1 a_0 + \cdots + f_k a_0^k \\ f_0 + f_1 a_1 + \cdots + f_k a_1^k \\ \cdots \\ f_0 + f_1 a_n + \cdots + f_k a_n^k \end{bmatrix} = \begin{bmatrix} 1 & a_0 & a_0^2 & \cdots & a_0^k \\ 1 & a_1 & a_1^2 & \cdots & a_1^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & a_n^2 & \cdots & a_n^k \end{bmatrix} \times \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_k \end{bmatrix} . \tag{1}$$

We denote the matrix by V and notice that the vector on the right side is the vector $\vec{f}$ of polynomial coefficients. This way we can rewrite equation (1) as follows:

$$\vec{y} = V\vec{f} \ .$$

This shows that the evaluation mapping between the coefficients $f_0, \ldots, f_n$ and evaluations $f(a_0), \ldots, f(a_n)$ of a polynomial is a linear tranformation determined by the matrix V.

3.2.2 Reconstructing the polynomial

If $k = n$ then the matrix V is a $(k+1) \times (k+1)$ square matrix. A matrix in this form is known as the Vandermonde matrix. It's determinant is equal to [Kil05, page 147]

$$\Delta(V) = \prod_{\substack{i,j \\ i>j}} (a_i - a_j) \ .$$

We need the evaluation mapping to be reversible and for this we need to show that the matrix V is invertible. A matrix is invertible, if it is regular that is, its determinant is invertible [Kil05, page 143]. We have defined the values of $a_0, \ldots, a_n$ to be distinct so the differences $(a_i - a_j)$ in the given sum are nonzero. To achieve that the product of the differences is nonzero it is enough to make sure that the ring has no zero divisors. For that reason we require from now on that $\mathbf{R}$ is a field, since fields have no zero divisors. With this assumption we ensure that $\Delta(V)$ is invertible and therefore V is invertible, if all values a_i are distinct. This in turn confirms that the transformation provided by V is also invertible and we can express f by using the inverse of V.

$$\vec{f} = V^{-1}\vec{y}$$

We will now show, how to reconstruct the polynomial f from its evaluations. We define vectors $\vec{e_i}$ as unit vectors in the form $\begin{bmatrix} e_0 & \ldots & e_n \end{bmatrix}$.

$$\begin{aligned} \vec{e_0} &= \begin{bmatrix} 1 & 0 & \ldots & 0 \end{bmatrix} \\ \vec{e_1} &= \begin{bmatrix} 0 & 1 & \ldots & 0 \end{bmatrix} \\ &\cdots \\ \vec{e_n} &= \begin{bmatrix} 0 & 0 & \ldots & 1 \end{bmatrix} \end{aligned}$$

Let $\vec{b_i}$ be such that

$$\vec{e_i} = V\vec{b_i}. \tag{2}$$

Because of the properties of V we showed earlier we can rewrite equation (2) and express $\vec{b_i}$ as follows.

$$\vec{b_i} = V^{-1}\vec{e_i} \ .$$

Noting that

$$\vec{y} = \sum_{i=0}^{n} y_i\vec{e_i}$$

we see that we can reconstruct $\vec{f}$ from evaluations as follows

$$\vec{f} = \sum_{i=0}^{n} V^{-1}y_i\vec{e_i} = \sum_{i=0}^{n} y_i\vec{b_i} \ .$$

It follows that we can reconstruct the coefficients of the polynomial f, if we have access to its evaluations at $n+1$ positions and the vectors $\vec{b_i}$ and therefore we have constructively proved the well-known Lagrange interpolation theorem.

Theorem 3.2 (Lagrange interpolation theorem)**.** *Let* $\mathbf{R}$ *be a field and* $a_0, \ldots, a_n$, $y_0, \ldots, y_n \in \mathbf{R}$ *so that all values* a_i *are distinct. Then there exists only one polynomial* f *over* $\mathbf{R}$ *so that* $\deg f \leq n$ *and* $f(a_i) = y_i$, $(i = 0, \ldots, n)$. □

The Lagrange interpolation polynomial can be computed as the sum

$$f(x) = y_0b_0(x) + \cdots + y_nb_n(x)$$

where the base polynomials $b_i(x)$ are defined as

$$b_i(x) = \prod_{\substack{j=0 \\ i\neq j}}^{n} \frac{(x - a_j)}{(a_i - a_j)} \ .$$

As one could expect, the Lagrange interpolation polynomial has a useful property—its base polynomials correspond to our vectors $\vec{b_i}$:

$$\mathbf{eval}(b_i) = \left[\begin{array}{ccc} b_i(a_0) & \ldots & b_i(a_n) \end{array} \right]^T$$

Since

$$b_i(a_j) = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases}$$

we see that

$$\mathbf{eval}(b_i) = \vec{e_i} \ .$$

We also have to handle the cases where $k \neq n$. We will reduce these cases to the $(k+1) \times (k+1)$ case observed before. First we consider the case where $n > k$. If we choose $k+1$ different values $l_0, \ldots, l_k \in \{0, \ldots, n\}$, then we obtain a virtual matrix V' by choosing rows $l_0, \ldots, l_k$ from the original matrix V:

$$V' = \begin{bmatrix} 1 & a_{l_0} & a_{l_0}^2 & \cdots & a_{l_0}^k \\ 1 & a_{l_1} & a_{l_1}^2 & \cdots & a_{l_1}^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_{l_k} & a_{l_k}^2 & \cdots & a_{l_k}^k \end{bmatrix} .$$

The square matrix V' is invertible as its determinant is nonzero, because it corresponds to the evaluation map at $[a_{l_0}, \ldots, a_{l_k}]$ and by showing that we have reached the previously observed and proved case. In the third case when $n < k$ we generate $k - n$ values $a_{n+1}, \ldots, a_k$ so that all values a_i are distinct. We use these new positions to add rows to the matrix V and get the virtual matrix V'

$$V' = \begin{bmatrix} 1 & a_0 & a_0^2 & \cdots & a_0^k \\ 1 & a_1 & a_1^2 & \cdots & a_1^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & a_n^2 & \cdots & a_n^k \\ 1 & a_{n+1} & a_{n+1}^2 & \cdots & a_{n+1}^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_k & a_k^2 & \cdots & a_k^k \end{bmatrix} .$$

This matrix V' is an invertible $(k+1) \times (k+1)$ square matrix that can replace V in the first observed case. Note that if $n < k$ the reconstruction of the polynomial is not unique and is determined by the choice of values $a_{n+1}, \ldots, a_k$. This gives us a guarantee that it is not possible to uniquely reconstruct the polynomial if there are not enough pairs of positions and evaluations available. Later we will use this property to prove privacy of the following secret sharing scheme.

3.3 Shamir's secret sharing scheme

We now describe Shamir's secret sharing scheme that is based on polynomial evaluations [Sha79]. We start by explaining the infrastructure of secret sharing. The central party is the dealer that performs share computation operations on input secrets and distributes the resulting shares to other parties. When the secret has to be reconstructed, the parties give their shares to the dealer, that can then combine the shares and retrieve the secret.

In Shamir's scheme shares are evaluations of a randomly generated polynomial. The polynomial f is generated in such a way that the evaluation $f(0)$ reveals

the secret value. If there are enough evaluations, the parties can reconstruct the polynomial and compute the secret. Algorithm 1 describes how shares are computated in Shamir's scheme.

Algorithm 1: Share computation algorithm for Shamir's scheme

Data: finite field $\mathbf{F}$, secret data $s \in \mathbf{F}$, threshold k, number of shares n
Result: shares $s_1, \ldots, s_n$
Set $f_0 = s$
Uniformly generate coefficients $f_1, \ldots, f_{k-1} \in \mathbf{F}$
Construct the polynomial $f(x) = f_0 + f_1 x + \cdots + f_{k-1} x^{k-1}$
Evaluate the polynomial: $s_i = f(i)$, $(i = 1, \ldots, n)$

Note that the indices of the shares start from one, as we cannot output $s_0 = f(0)$, because it is the secret value. The resulting shares $s_1, \ldots, s_n$ can be distributed to their holders. If the original value needs to be retrieved, we need a subset of at least k shares. Note that it is important to store the index i together with the share s_i, because it is later needed for reconstruction.

The classical algorithm of Shamir's scheme reconstructs the whole polynomial, whereas we describe versions optimised for reconstructing only the secret $f(0) = s$. We only need to compute $f(0)$ so for our purposes we can simplify the base polynomials $b_i(x)$ as follows:

$$\beta_i = b_i(0) = \prod_{\substack{j=1 \\ i \neq j}}^{k} \frac{(-a_j)}{(a_i - a_j)}. \tag{3}$$

If the shares are computed using Shamir's scheme then algorithm 2 retrieves the secret value s.

Algorithm 2: Share reconstruction algorithm for Shamir's scheme

Data: finite field $\mathbf{F}$, shares $s_{t_1}, \ldots, s_{t_k} \in \mathbf{F}$ where $t_j \in \{1, \ldots, n\}$ are distinct indices
Result: secret data s
compute the reconstruction coefficients β_i according to equation (3)
compute $f(0) = s_{t_1}\beta_{t_1} + \cdots + s_{t_k}\beta_{t_k}$
Return $s = f(0)$

Theorem 3.3. *Shamir's secret sharing scheme is correct and private in the sense of Definition 3.1.*

Proof. Correctness follows directly from the properties of Lagrange interpolation. The indices and shares are really positions and values of the polynomial and thanks

to the properties of Lagrange interpolation they uniquely determine this polynomial. The algorithm for finding the polynomial is also efficient, as it consists of polynomial evaluations.

Note that we have considered secret polynomials of degree n. To continue doing this we must note, that such polynomials are achieved in a $(n+1)$-out-of-$(n+1)$ instantiation of Shamir's secret sharing scheme. For this reason the following proof is given for a $(n+1)$-out-of-$(n+1)$ secret sharing scheme.

To prove security of the scheme we show that the shares are from a uniform distribution and revealing n shares to an adversary does not give it any information about the secret value. We do this by creating a situation where one share is missing at reconstruction. We transform the matrix V by removing any one row from it. Then we separate the first column of ones to create the vector $\mathbf{1}$ that consists of n ones and get a $n \times n$ matrix A. We can now express the vector of shares $\vec{y'}$ as follows:

$$\vec{y'} = \mathbf{1}f_0 + A\vec{f'} \tag{4}$$

where

$$\vec{f'} = \begin{bmatrix} f_1 & \dots & f_n \end{bmatrix}^T \quad .$$

The vector $\vec{y'}$ represents n shares and the vector $\mathbf{1}f_0 = \begin{bmatrix} f_0 & \dots & f_0 \end{bmatrix}^T$ contains the secret values f_0. We notice that $A\vec{f'}$ is uniform, because values f_i in the vector $\vec{f'}$ are coefficients of the secret polynomial and were generated from a uniform distribution. Because the values a_i are different nonzero values, the determinant of A is nonzero and the matrix A is invertible. It follows that A defines a bijective transformation—for any vector $\vec{f'}$ the matrix A transforms it to exactly one vector. Hence, $A\vec{f'}$ has a uniform dstribution, since $\vec{f'}$ has a uniform distribution.

Using this property we can show that $\vec{y'}$ is from a uniform distribution, because if we add a vector $A\vec{f'}$ that has uniformly distributed values to a non-random vector $\mathbf{1}f_0$ we get a vector from a uniform distribution. The equation (4) shows us that combinations of n shares are from a uniform distribution and therefore give no information about the secret value. This means the adversary cannot do better than guess the missing share, but all guesses are equally probable.

The proof for a $(k+1)$-out-of-$(n+1)$ is reduced to the $(n+1)$-out-of-$(n+1)$ case as follows. We note that in a $(k+1)$-out-of-$(n+1)$ scheme $\deg f = k$, which means that

$$\vec{f'} = \begin{bmatrix} f_1 & \dots & f_k \end{bmatrix}^T \quad .$$

The matrix V contains $n+1$ columns and $n+1$ rows from which we select k rows to form a $(k+1) \times k$ matrix V'. This simulates the situation when we have only k shares available at reconstruction and reduces the problem to the already proved $(n+1)$-out-of-$(n+1)$ case, because we can now separate the column of ones from V' and construct the $k \times k$ matrix A. This allows us to follow the same discussion

as in the $(n+1)$-out-of-$(n+1)$ case and prove that k shares reveal nothing about the secret in a $(k+1)$-out-of-$(n+1)$ scheme. □

3.4 Secure computation with shares

We will now show what can be done with the shares once they have been distributed. We will investigate the possibility of using the homomorphic property of the secret sharing scheme to perform operations with the shares. In the following assume that a k-out-of-n threshold scheme is used. Assume that we have n parties $P_1, \ldots, P_n$ and the dealer gives each one of them a share according to its index.

Addition. Assume that we have shared values $[\![u]\!] = [u_1, \ldots, u_n]$ and $[\![v]\!] = [v_1, \ldots, v_n]$. Because the evaluation mapping is a linear transformation, we can add the shares of $[\![u]\!]$ and $[\![v]\!]$ to create a shared value $[\![w]\!]$ so that $u + v = w$. Each party k has to run the protocol given in Algorithm 3 to add two shared values.

Algorithm 3: Protocol for adding two Shamir shares for node k

Data: shares u_k and v_k
Result: share w_k that represents the sum of $[\![u]\!]$ and $[\![v]\!]$
`Round 1`
$w_k = u_k + v_k$

Multiplication with a scalar. Assume that we have a shared value $[\![u]\!] = [u_1, \ldots, u_n]$ and a public value t. Again, thanks to the linear transformation property of the evaluation mapping we can multiply the shares u_i with t so that the resulting shares represent the value $[\![w]\!] = t[\![u]\!]$. Algorithm 4 shows the protocol for multiplication a share value by a scalar.

Algorithm 4: Protocol for multiplying Shamir shares by a scalar value for node k

Data: shares u_k and a public value t
Result: share w_k that represents the value of $t[\![u]\!]$
`Round 1`
$w_k = tu_k$

Multiplication. Assume that we have shared values $[\![u]\!] = [u_1, \ldots, u_n]$ and $[\![v]\!] = [v_1, \ldots, v_n]$. Share multiplication, unfortunately, cannot be solved with the linear property of the transformation, as multiplying two polynomials with the same degree gives a polynomial with double the degree of the source polynomials. This means that we must use a k-out-of-n threshold scheme where $2k \leq n$ and the polynomials must have a degree of at most $2k$. By multiplying the respective shares, the miners actually compute a share that represents the polynomial storing

the the product of the secrets. However, we must reconstruct the secret stored in the product polynomial and reshare it to make further multiplications possible. Otherwise, the multiplication of the product polynomial with another one will give us a polynomial with a degree larger than n and we cannot reconstruct the secret from such polynomials anymore.

We note that we can precompute the values of the optimised base polynomials β_i needed in the protocol by using equation (3) on page 26. This requires each node to know its number and also how many other nodes there are, but that is a reasonable assumption. Algorithm 5 gives the complete protocol for multiplying Shamir shares.

Algorithm 5: Protocol for multiplying two Shamir shares for node i

Data: shares u_i and v_i, precomputed value β_i
Result: share w_i that represents the value of $[\![u]\!][\![v]\!]$
`Round 1`
  $z_i = u_i v_i \beta_i$
  Share z_i to $z_{i_1}, \ldots, z_{i_n}$ using the same scheme as the dealer uses
  Send to each other node $P_l, i \neq l$ the share z_{i_l}
`Round 2`
  Receive shares $z_{ji}, j \neq i$ from other nodes
  $w_i = z_{i_i} + \sum_{\substack{j=1 \\ j \neq i}}^{n} z_{ji}$

Theorem 3.4. *The share multiplication protocol is correct.*

Proof. The protocol is based on the observation that in Shamir's secret sharing scheme the value is stored in the constant term of the polynomial and if we multiply two polynomials, the constant term of the resulting polynomial stores the product of secrets. In the first round of the protocol we multiply our shares of the two polynomials and use the precomputed base polynomials $\beta_i(0)$ from (3) to compute an element of the sum in Shamir's reconstruction algorithm shown in Algorithm 2. Basically, each miner computes its share of the reconstruction formula for the product and then distributes it to the other miners as shares. Each miner can add the received shares because the scheme is additive. As a result each miner has a share of the product of u and v. □

Theorem 3.5. *The share multiplication protocol is perfectly secure in a semi-honest model with three parties.*

Proof. We present a sketch for the security proof in a scenario with three parties for easier understanding. The proof can be extended to support more than three

parties. The security of the multiplication protocol is based on the security of local computations and the security of Shamir's scheme that was proved in Theorem 3.3.

We prove that the protocol is secure by showing that the messages received by one party are uniformly distributed. Then we can build the necessary simulator using the technique shown in Theorem 2.1. We show security for one party as the other cases follow trivially because the protocol is symmetrical.

The party P_1 receives two values: z_{21} and z_{31} that are shares of z_2 and z_3 respectively. Since Shamir's scheme is private, P_1 can learn nothing about values z_2 and z_3 from these shares. The values z_{21} and z_{31} are also independent as they have been formed by two independently executed share distributions performed by P_2 and P_3. □

3.5 Generalised secret sharing

We showed how to build a secret sharing scheme on a linear transformation representing polynomial evaluations. We now go further and discuss connections between secret sharing and classical coding theory. Figure 9 illustrates the similarity of coding and secret sharing. Both transform an input value to an intermediate format and later reconstruct it to retrieve the original value.

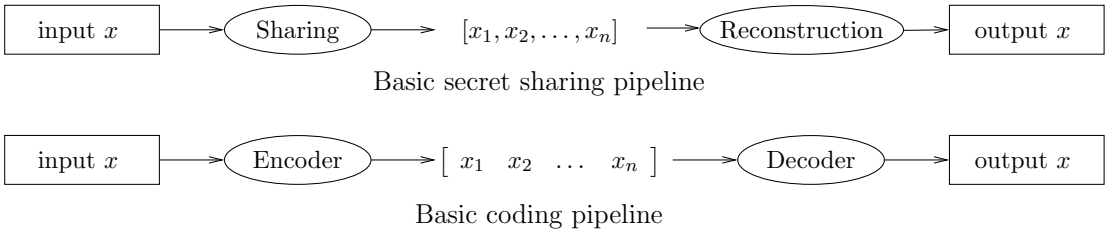

Figure 9: Comparison of the basic pipelines of secret sharing and coding

Standard coding schemes have encoding and decoding procedures that correspond to share computation and reconstruction operations in secret sharing. The following scheme represents a generic randomised coding scheme.

$$\mathsf{Encode}(x_0) = A \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_k \end{bmatrix}, \text{ where } x_1, \ldots, x_k \leftarrow \mathbb{Z}_p$$

$$\mathsf{Decode}(y_1, \ldots, y_n) = \langle \beta, y \rangle = \beta^T y = \sum_{i=1}^{n} \beta_i y_i$$

where $\beta = [\beta_1, \ldots, \beta_n]$ is the reconstruction vector.

For example, if A is a Vandermonde matrix then the vector $\vec{\beta}$ corresponds to the evaluations of base polynomials $\beta_i = b_i(0)$ in the Lagrange interpolation formula.

The output of the encoding algorithm can be interpreted as shares. In secret sharing the access structure specifies which parties can reconstruct the value. The corresponding operation in coding theory is recovering from deletion errors. In coding theory we may learn something about the original value even if some codewords are missing. Secret sharing requires an all-or-nothing approach where having less than the sufficient number of shares provides no information about the original value,

Coding theory also allows recovering from errors and detecting the codeword that was damaged during transmission. Similarly, verifiable secret sharing allows us to detect if a share has been modified and identify the dishonest party in the system. In both cases there are limits to how many errors there can be before we cannot decode the value from codewords or detect the cheater in a secret sharing scheme. It has been shown that secret sharing is a special case of linear coding [MS81].

To give an example of a verifiable secret sharing scheme we slightly modify the generic randomised coding scheme presented above. For an input value x_0 let us define the input vector $\vec{x}$ as follows:

$$\vec{x} = \left[\begin{array}{ccccccc} x_0 & \dots & x_k & x_{k+1} & \dots & x_l \end{array} \right]^T,$$

where $x_1, \dots, x_k \leftarrow \mathbb{Z}_p$ and $x_{k+1}, \dots, x_l = 0$. Now we can define the scheme.

$\mathsf{Encode}(x_0) = A\vec{x}$

$\mathsf{Decode}(y_1, \dots, y_n) = \langle b, y \rangle$

$$\mathsf{Verify}(y_1, \dots, y_n) = \begin{cases} true, & \text{if } x_{k+1}, \dots, x_l \text{ in the restored } \vec{x} \text{ are zeroes,} \\ false, & \text{otherwise.} \end{cases}$$

To reconstruct the original value after the adversary has corrupted one or more shares is possible, if we have enough honest parties, i.e., uncorrupted shares. We note, that if all parties are honest, any set of shares with cardinality equal or above the threshold value will be able to reconstruct the same polynomial. Now, if some parties provide corrupted shares we will have to reconstruct all possible polynomials to see, which shares make the polynomials mismatch with the correct ones. It is easy to see, that this task is not trivial and requires a lot of reconstructing and verification of secrets together with the analysis of the achieved results.

4 A framework for secure computations

4.1 Introduction and goals

In this chapter we give a detailed description of our solution for performing secure data aggregation. First, we list our design goals considered when deciding the features of the system. We wanted to build a theoretical framework for designing secure data processing applications and our main design goal was efficiency and ease of use. We also wanted to build a programming platform that could be used to make development of secure computations software easier. Thus we had to create a the theoretical framework that provides us with security guarantees without sacrificing real-world performance.

We start describing our framework by listing the types of parties and explaining their model of communication. Then we build the mathematical foundations for the protocols by describing the value space and the secret sharing scheme. Based on the infrastructure and the mathematical concepts we assess the security model of the system. Finally, we present protocols for basic operations like adding, multiplying and comparison along with their proofs of security.

4.2 Mathematical foundations

We require all elements of the data to be values in $\mathbb{Z}_{2^{32}}$. Note that $\mathbb{Z}_{2^{32}}$ is not a finite field with respect to integer addition and multiplication. We prefer this approach, because there are platforms and programming languages where operations in $\mathbb{Z}_{2^{32}}$ are natively available. That is, the computer natively performs arithmetics mod 2^{32}. This allows us to have an implementation with efficient computations and rapid development.

We use the simple and efficient n-out-of-n additive secret sharing scheme. Secret values and shares are stored in $\mathbb{Z}_{2^{32}}$ which allows for fast secret sharing operations. Algorithm 6 describes how shares are computed in the additive scheme.

Algorithm 6: Share computation algorithm for the additive scheme

Data: secret data $s \in \mathbb{Z}_{2^{32}}$, number of shares n
Result: shares $s_1, \ldots, s_n$
Uniformly generate values $s_1, \ldots, s_{n-1} \in \mathbb{Z}_{2^{32}}$
$s_n = s - s_1 - \cdots - s_{n-1}$

Note, that Algorithm 6 can be rearranged so that any share s_i is computed by subtraction in the final step. This is useful to know if we have to show that any $n - 1$ shares are uniformly generated.

Algorithm 7 shows how to reconstruct the original value from additive shares.

Algorithm 7: Share reconstruction algorithm for the additive scheme

Data: shares $s_1, \ldots, s_n \in \mathbb{Z}_{2^{32}}$
Result: secret data s
$s = s_1 + \cdots + s_n$

We now give the security proof for the additive scheme.

Theorem 4.1. *The secret sharing scheme is correct and private in the sense of Definition 3.1.*

Proof. Correctness is trivial, as the reconstruction algorithm is both efficient and determines the secret uniquely. To show privacy we note, that the share generation process shown in Algorithm 6 assures, that values $s_1, \ldots, s_n$ are uniformly chosen from the set $S(s) = \{(s_1, \ldots, s_n) : s_1 + \ldots, +s_n = s \bmod 2^32\}$. Now observe, that due to symmetry we can sample $S(s)$ by choosing a value $1 \leq i \leq n$, then generating $s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n \leftarrow \mathbb{Z}_N$ and finally computing $s_i = s - s_1 - \cdots - s_{i-1} - s_{i+1} - \cdots - s_n$. □

Theorem 4.2. *For any two shared values* $[u]$ *and* $[v]$ *and a scalar value* $r \in \mathbb{Z}_{2^{32}}$ *the following conditions hold:*

i ***Additivity:*** $[u + v] = [(u_1 + v_1), \ldots, (u_n + v_n)]$.

i ***Multiplicativity w.r.t. scalar values:*** $r[u] = [ru_1, \ldots, ru_n]$.

Both properties follow directly from the construction of the scheme. It follows that this secret sharing scheme is $(+, +)$-homomorphic with respect to addition and $(\times, \times)$-homomorphic with respect to multiplication by scalar. Unfortunately, it is not multiplicative, as it is easy to construct counter-examples that show that $[uv] \neq [(u_1v_1), \ldots, (u_nv_n)]$. Due to this restriction we have to solve multiplication differently. We give a suitable protocol later in Section 4.5.3.

4.3 The infrastructure

We will now describe the infrastructure of our solution. We have three central computing parties M_1, M_2 and M_3 called *miners*. Their job is to perform computations and run data mining algorithms on the data. Each miner is aware of its index—1, 2 or 3. The other nodes in the system are controllers $C_1, \ldots, C_n$ that send data to the miners and request analysis results from them. There is no theoretical limit to

the number of controllers n. Controllers may have different functionalities—some might only provide data while the others may request analysis results.

We have communication channels between all the miners so they can run secure multy-party computation protocols. It is assumed that all parties in the system belong to a a public key infrastructure so all nodes in the system are capable of running public key encryption and signing. This is not necessary for communication between the parties but it allows us to receive encrypted data from arbitrary clients. Symmetrical encryption and authentication codes are used in the traffic between parties to achieve security described in the cryptographic model of communication.

Figure 10 illustrates the deployment of miners and their communication with both the controller and the other miners.

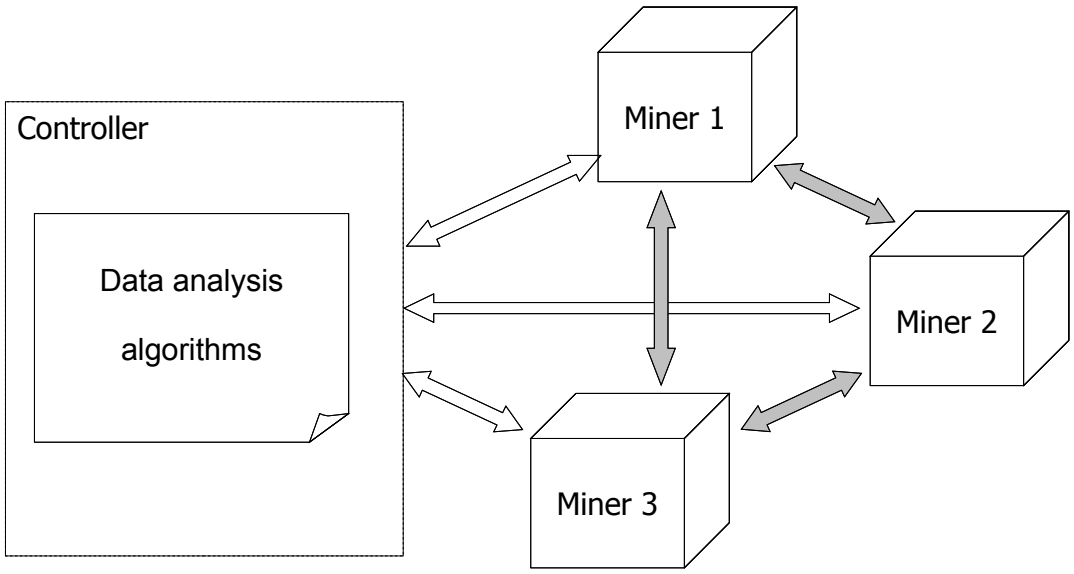

Figure 10: The communication channels between miners and the controller

Each controller must connect to all three miners to perform operations. The reason for this is security—data needs to be secret-shared as soon as possible to minimize the risk of compromising it. Each controller acts as a dealer in the context of a secret sharing scheme. If data needs to be sent to the miners, the controller distributes it into shares and sends each share to one miner. As a result, the data is separated into three parts, all of which must be combined to recreate the original values.

We now describe a secure method for non-interactive data collection that requires a central encrypted data repository. The repository is a public database accessible to all miners and controllers—the controllers write to it and the miners

read from it. If a controller has new data to send to the miners it forms an $p \times q$ matrix $\mathcal{D}$ which contains the data. Then it generates three AES keys k_1, k_2, k_3 and encrypts them by using the miners' public keys. The encrypted keys are sent to the respective miners M_1, M_2 and M_3.

Now let $c : \mathbb{Z}_p \times \mathbb{Z}_q \to \mathbb{Z}_{pq}$ be an injective transformation. For each matrix element $\mathcal{D}_{ij}$, $i = 1, \ldots, p$, $j = 1, \ldots, q$, the controller computes three values $e_{1,ij}, e_{2,ij}, e_{3,ij}$ by encrypting the value $c(i, j)$ with each of the three generated keys k_1, k_2 and k_3. Then it publishes the values $s_{ij} = \mathcal{D}_{ij} + e_{1,ij} + e_{2,ij} + e_{3,ij}$ by storing them in the central repository.

The miners retrieve the values s_{ij} and use the received AES key to compute the values $e_{*,ij}$. Each miner now knows s_{ij} and one of the values $e_{*,ij}$. Based on this they compute additive shares of $\mathcal{D}_{ij}$ as follows:

(i) The first miner's share is $(S_{ij} - e_{1,ij})$,

(ii) The second miner's share is $(-e_{2,ij})$,

(iii) The third miner's share is $(-e_{3,ij})$.

It is easy to see the the shares represent the value $\mathcal{D}_{ij}$.

A note about implementing this scheme: since AES provides us with values in $\mathbb{Z}_{2^{128}}$ while our data might be represented with less bits, we have to either lose the superfluous bits from AES or use each AES block for sharing more than one value. This is easy to do, if the values are powers of two, for example, elements in $\mathbb{Z}_2$, $\mathbb{Z}_{2^8}$ and so on.

This structure minimises source data exposure. In an ideal scenario, we run a survey electronically and give each person who is answering questions a digital survey built upon a controller node. This way no-one but the author can see the answers, because the miners only return aggregation results. The miners will have to make use of microdata protection methods to make sure that aggregation results do not reveal too much information about the original data.

The miners can run a number of basic operations like arithmetic and data manipulation. More complex algorithms are built from these basic operations. When a controller wants the miners to run an algorithm, it sends the program code to the miners in an assembly-like format. The miner can verify the code to ensure that it does not compute anything that would compromise privacy. The miners then execute the code by running each basic operation until the final results are computed. The controllers see only the final results of the computation, intermediate results are kept by the miners for security reasons. To ensure the authenticity of the code, we require the controllers to sign the code they send to the miners.

4.4 Security model

We have three miners and any number of controllers. The controllers are sources of data and consumers of data mining results. If all controllers collaborated, they could put together the original dataset because they have access to the source data. We assume that the data providers are concerned with the privacy of their data, otherwise no method or technology can prevent them from distributing it freely. In our security goals we concentrate on the miners that in real-life scenarios hold all of the data and have lots of time to work on it.

We use the additive secret sharing scheme to distribute data between the three miners. Input data is distributed into three shares and each party has one share of each original value. Note that we can prevent malicious or unfair share distribution by the controller by using share conversion protocols to retain the secret value but change the shares. To enforce a range on input shares we may convert them by removing the extra bits. For example, for one-bit inputs we will just use the first bit of each share as the additive scheme provides correct reconstruction for $\mathbb{Z}_2$. Since our protocols require uniformity in $\mathbb{Z}_{2^{32}}$ we will then have to convert these shares back to shares in $\mathbb{Z}_{2^{32}}$. This is still better and more efficient than requiring zero-knowledge proofs from the clients.

We have three miners in a secure multiparty computation setting with bidirectional communication channels. We perform computations synchronously—basic operations are run simultaneously on all miners and we also expect basic operations to complete before a certain time bound. We allow all parties to be semi-honest—they have to follow the protocol, but they can be curious about the data. We provide security only if the adversary corrupts at most one party of the three which means that no two parties can collaborate.

To prove security of our protocols we have to show that all messages received by a any fixed party are uniformly distributed and independent. This allows us to prove that if the adversary corrupts one party, it is sent only uniformly distributed values that give it no information about the secret values. With the properties of uniformity and indepence it is trivial to build the necessary simulators which are required for perfect security of the protocols.

We have designed the protocols to be symmetrical with minor exceptions. Mostly each miner performs the same operations, just with different values. We use this property in security proofs to show that one miner receives only uniformly distributed independent messages and reuse this proofs for the other two miners. Our protocols use each other as sub-protocols. Such compositions of protocols are also perfectly secure [Dam02, page 8].

Note that at the end of our protocols each party holds an additive share of the final result. These shares must be from a uniform distribution, because then, according to corollary 2.1 we can publish them without compromising security.

This means that after completing the protocol the parties can send their shares to a controller that can reconstruct results of the computation. Note that this is not necessary, if the shares are only used by the miners later on and this could save us from sending some messages in the presented protocols. However, to maintain generality the presented protocols conform to this requirement.

4.5 Protocols for basic operations

4.5.1 Prerequisites and definitions

We now describe the protocols for performing basic arithmetic operations. Each operation works on shared values and outputs also shared values. At no point are the original secret values reconstructed during the execution of protocols. The input parameters are described together with each algorithm.

We present these protocols as algorithms run simultaneously by three miners M_1, M_2 and M_3. For easier reference we call them Alice, Bob and Charlie, respectively. We use the following common notations in protocol descriptions. We use $[\![x]\!]$ to denote a value that is shared between the miners using the simple additive secret sharing scheme. The respective shares are x_A, x_B and x_C where the indices specify the party holding the share. A stands for Alice, B for Bob and C for Charlie. Messages in the protocol are denoted var_{xy} where $x \in \{1, 2, 3\}$ is the index of the sending party and $y \in \{1, 2, 3\}$, $y \neq x$ is the index of the receiving party. Note, that var may consist of more than one letter and still denote a single value and not a multiplication. In the case of a possible ambiguity we use the $\cdot$ operator to specify multiplication.

4.5.2 Addition and multiplication by scalar

Given input values $[\![u]\!]$ and $[\![v]\!]$, we want to compute $[\![u + v]\!]$. We showed that the secret sharing scheme is homomorphic with respect to addition and multiplication by scalar so we can do these operations locally in one round. Algorithm 8 gives the protocol for adding two additively shared values.

Algorithm 8: Protocol for adding two additive shares

Data: values $[\![u]\!]$ and $[\![v]\!]$
Result: the sum $[\![u + v]\!]$
`Round 1`
Alice computes $(u + v)_A = u_A + v_A$
Bob computes $(u + v)_B = u_B + v_B$
Charlie computes $(u + v)_C = u_C + v_C$

Given input value $[u]$ and a scalar r we want to compute $r[u]$. Algorithm 9 shows the protocol for multiplying additively shared data by a scalar value.

Algorithm 9: Protocol for multiplying an additive shares by a scalar value

Data: value $[u]$ and a scalar r
Result: the product $[u']$
`Round 1`
Alice computes $u'_A = r \cdot u_A$
Bob computes $u'_B = r \cdot u_B$
Charlie computes $u'_C = r \cdot u_C$

4.5.3 Multiplication

Given input values $[u]$ and $[v]$ we want to compute $[uv]$. Since we showed that the secret sharing scheme is not homomorphic with respect to multiplication, we need to build a protocol to perform the computation. The idea behind the algorithm was introduced by Du and Atallah [DA00, page 11]. Namely, assume that two parties P_1 and P_2 want to multiply values x_1 and x_2 without anyone else knowing x_1 or x_2. They notice, that

$$(x_1 + \alpha_1)(x_2 + \alpha_2) = x_1x_2 + x_1(x_2 + \alpha_2) + x_2(x_1 + \alpha_1) - \alpha_1\alpha_2,$$

which allows us to express the product x_1x_2 as follows:

$$x_1x_2 = (x_1 + \alpha_1)(x_2 + \alpha_2) - x_1(x_2 + \alpha_2) - x_2(x_1 + \alpha_1) + \alpha_1\alpha_2 \ . \tag{5}$$

Now assume that P_1 has uniformly chosen $\alpha_1 \leftarrow \mathbb{Z}_N$ and P_2 has independently uniformly chosen $\alpha_2 \leftarrow \mathbb{Z}_N$. Then the sums $x_1+\alpha_1$ and $x_2+\alpha_2$ are also independent and from a uniform distribution which means that they can be sent as protocol messages without leaking data. However, P_1 and P_2 still cannot compute x_1x_2 because they do not have the product $\alpha_1\alpha_2$. For completing the protocol they need a third party P_3 that generates the values α_1 and α_2 and therefore knows $\alpha_1\alpha_2$. By combining the available values the parties can now compute shares of x_1x_2. The respective protocol consists of the following steps:

1. The party P_3 uniformly generates $\alpha_1, \alpha_2 \leftarrow \mathbb{Z}_N$ and sends α_1 to P_1 and α_2 to P_2.

2. P_1 computes $x_1 + \alpha_1$ and sends the result to P_2. At the same time P_2 computes $x_2 + \alpha_2$ and sends the result to P_1.

3. Now the parties have enough information to compute shares of the product x_1x_2:

   - P_1 computes its share $p_1 = (x_1 + \alpha_1)(x_2 + \alpha_2) - x_1(x_2 + \alpha_2)$
   - P_2 computes its share $p_2 = -x_2(x_1 + \alpha_1)$
   - P_3 computes its share $p_3 = \alpha_1\alpha_2$

This protocol is trivially correct as the shares p_1, p_2 and p_3 add up to the product x_1y_1 as shown in equation (5) and it is also easy to show that it is perfectly secure. In the first step P_1 and P_2 receive independent uniformly generated values α_1 and α_2. In the second step P_1 receives $x_2 + \alpha_2$ that is uniformly distributed and independent from the values P_1 has received up to then. Similarly we can show that the messages received by P_2 reveal nothing about the computed product.

Now we notice that

$$\begin{aligned} uv &= (u_A + u_B + u_C)(v_A + v_B + v_C) \\ &= u_Av_A + u_Av_B + u_Av_C + u_Bv_A + u_Bv_B + u_Bv_C + u_Cv_A + u_Cv_B + u_Cv_C. \end{aligned} \tag{6}$$

This means that to compute uv we have to compute the products of individual shares u_iv_j. Three of them—u_Av_A, u_Bv_B and u_Cv_C—can be computed locally and we can compute the others by using the above protocol. We know that running perfectly secure protocols in parallel gives us a perfectly secure protocol [Dam02] so we can compose our share multiplication protocol out of six instances of the above protocol to securely compute the needed products in equation (6).

Note that we also need the resulting shares to be from a uniform distribution. For that each node uniformly generates an additional value and sends it to the next node in the numerical ordering and the last node sends the value to the first node. Each party then subtracts the sent value from its share and adds the received value. Algorithm 10 gives the complete protocol for multiplying two additively shared values.

Algorithm 10: Share multiplication for additive shares

Data: values $[u]$ and $[v]$
Result: the product $[uv]$
`Round 1`
Alice generates $r_{12}, r_{13}, s_{12}, s_{13}, t_{12} \leftarrow \mathbb{Z}_{2^{32}}$
Bob generates $r_{23}, r_{21}, s_{23}, s_{21}, t_{23} \leftarrow \mathbb{Z}_{2^{32}}$
Charlie generates $r_{31}, r_{32}, s_{31}, s_{32}, t_{31} \leftarrow \mathbb{Z}_{2^{32}}$
All values $*_{ij}$ are sent from M_i to M_j
`Round 2`
Alice computes
$\hat{a}_{12} = u_A + r_{31}$, $\hat{b}_{12} = v_A + s_{31}$, $\hat{a}_{13} = u_A + r_{21}$, $\hat{b}_{13} = v_A + s_{21}$
Bob computes
$\hat{a}_{23} = u_B + r_{12}$, $\hat{b}_{23} = v_B + s_{12}$, $\hat{a}_{21} = u_B + r_{32}$, $\hat{b}_{21} = v_B + s_{32}$
Charlie computes
$\hat{a}_{31} = u_C + r_{23}$, $\hat{b}_{31} = v_C + s_{23}$, $\hat{a}_{32} = u_C + r_{13}$, $\hat{b}_{32} = v_C + s_{13}$
All values $*_{ij}$ are sent from M_i to M_j
`Round 3`
Alice computes:
$c_A = u_A\hat{b}_{21} + u_A\hat{b}_{31} + v_A\hat{a}_{21} + v_A\hat{a}_{31} - \hat{a}_{12}\hat{b}_{21} - \hat{b}_{12}\hat{a}_{21} + r_{12}s_{13} + s_{12}r_{13} - t_{12} + t_{31}$
$uv_A = c_A + u_A v_A$
Bob computes
$c_B = u_B\hat{b}_{32} + u_B\hat{b}_{12} + v_B\hat{a}_{32} + v_B\hat{a}_{12} - \hat{a}_{23}\hat{b}_{32} - \hat{b}_{23}\hat{a}_{32} + r_{23}s_{21} + s_{23}r_{21} - t_{23} + t_{12}$
$uv_B = c_B + u_B v_B$
Charlie computes
$c_C = u_C\hat{b}_{13} + u_C\hat{b}_{23} + v_C\hat{a}_{13} + v_C\hat{a}_{23} - \hat{a}_{31}\hat{b}_{13} - \hat{b}_{31}\hat{a}_{13} + r_{31}s_{32} + s_{31}r_{32} - t_{31} + t_{23}$
$uv_C = c_C + u_C v_C$

At the end of the protocol each party has a share of the product $[uv]$. The computation requires three rounds and a total of 27 messages.

Theorem 4.3. *The share multiplication protocol is correct.*

Proof. Formally, the correctness of the protocol can be shown by comparing two expressions, uv and $uv_A + uv_B + uv_C$. First, we evaluate the shares of the product $[\![uv]\!]$:

$$
\begin{aligned}
uv_A &= c_A + u_A v_A \\
&= u_A\hat{b}_{21} + u_A\hat{b}_{31} + v_A\hat{a}_{21} + v_A\hat{a}_{31} - \hat{a}_{12}\hat{b}_{21} - \hat{b}_{12}\hat{a}_{21} + r_{12}s_{13} + s_{12}r_{13} - t_{12} + t_{31} \\
&\quad + u_A v_A \\
&= u_A v_B + u_A s_{32} + u_A v_C + u_A s_{23} + v_A u_B + v_A r_{32} + v_A u_C + v_A r_{23} \\
&\quad - u_A v_B - u_A s_{32} - r_{31} v_B - r_{31} s_{32} - v_A u_B - v_A r_{32} - s_{31} u_B - s_{31} r_{32} \\
&\quad + r_{12}s_{13} + s_{12}r_{13} - t_{12} + t_{31} + u_A v_A \\
&= u_A v_C + u_A s_{23} + v_A u_C + v_A r_{23} - r_{31} v_B - r_{31} s_{32} - s_{31} u_B - s_{31} r_{32} \\
&\quad + r_{12}s_{13} + s_{12}r_{13} - t_{12} + t_{31} + u_A v_A
\end{aligned}
$$

$$
\begin{aligned}
uv_B &= c_B + u_B v_B \\
&= u_B\hat{b}_{32} + u_B\hat{b}_{12} + v_B\hat{a}_{32} + v_A\hat{a}_{12} - \hat{a}_{23}\hat{b}_{32} - \hat{b}_{23}\hat{a}_{32} + r_{23}s_{21} + s_{23}r_{21} - t_{23} + t_{12} \\
&\quad + u_B v_B \\
&= u_B v_C + u_B s_{13} + u_B v_A + u_B s_{31} + v_B u_C + v_B r_{13} + v_B u_A + v_B r_{31} \\
&\quad - u_B v_C - u_B s_{13} - r_{12} v_C - r_{12} s_{13} - v_B u_C - v_B r_{13} - s_{12} u_C - s_{12} r_{13} \\
&\quad + r_{23}s_{21} + s_{23}r_{21} - t_{23} + t_{12} + u_B v_B \\
&= u_B v_A + u_B s_{31} + v_B u_A + v_B r_{31} - r_{12} v_C - r_{12} s_{13} - s_{12} u_C - s_{12} r_{13} \\
&\quad + r_{23}s_{21} + s_{23}r_{21} - t_{23} + t_{12} + u_B v_B
\end{aligned}
$$

$$
\begin{aligned}
uv_C &= c_C + u_C v_C \\
&= u_C\hat{b}_{13} + u_C\hat{b}_{23} + v_C\hat{a}_{13} + v_A\hat{a}_{23} - \hat{a}_{31}\hat{b}_{13} - \hat{b}_{31}\hat{a}_{13} + r_{31}s_{32} + s_{31}r_{32} - t_{31} + t_{23} \\
&\quad + u_C v_C \\
&= u_C v_A + u_C s_{21} + u_C v_B + u_C s_{12} + v_C u_A + v_C r_{21} + v_C u_B + v_C r_{12} \\
&\quad - u_C v_A - u_C s_{21} - r_{23} v_A - r_{23} s_{21} - v_C u_A - v_C r_{21} - s_{23} u_A - s_{23} r_{21} \\
&\quad + r_{31}s_{32} + s_{31}r_{32} - t_{31} + t_{23} + u_C v_C \\
&= u_C v_B + u_C s_{12} + v_C u_B + v_C r_{12} - r_{23} v_A - r_{23} s_{21} - s_{23} u_A - s_{23} r_{21} \\
&\quad + r_{31}s_{32} + s_{31}r_{32} - t_{31} + t_{23} + u_C v_C
\end{aligned}
$$

Now we combine the shares to verify that they reconstruct to the product uv.

$$
\begin{aligned}
uv_A + uv_B + uv_C \\
&= u_A v_C + u_A s_{23} + v_A u_C + v_A r_{23} - r_{31} v_B - r_{31} s_{32} - s_{31} u_B - s_{31} r_{32} \\
&\quad + r_{12} s_{13} + s_{12} r_{13} - t_{12} + t_{31} + u_A v_A + u_B v_A + u_B s_{31} + v_B u_A + v_B r_{31} \\
&\quad - r_{12} v_C - r_{12} s_{13} - s_{12} u_C - s_{12} r_{13} + r_{23} s_{21} + s_{23} r_{21} - t_{23} + t_{12} + u_B v_B \\
&\quad + u_C v_B + u_C s_{12} + v_C u_B + v_C r_{12} - r_{23} v_A - r_{23} s_{21} - s_{23} u_A - s_{23} r_{21} \\
&\quad + r_{31} s_{32} + s_{31} r_{32} - t_{31} + t_{23} + u_C v_C \\
&= u_A v_A + u_A v_B + u_A v_C + u_B v_A + u_B v_B + u_B v_C + u_C v_A + u_C v_B + u_C v_C \\
&= (u_A + u_B + u_C)(v_A + v_B + v_C) = uv.
\end{aligned}
$$

□

Theorem 4.4. *The share multiplication protocol is perfectly secure.*

Proof. Since the protocol is symmetrical, we can prove that in each round the messages received by P_1 are independent and from a uniform distribution and the other cases follow trivially. In the first round, P_1 receives four uniformly and independently generated values r_{21}, r_{31}, s_{21} and s_{31} that satisfy the security requirements by construction.

In the second round, P_1 receives the sums $\hat{a}_{21}$, $\hat{a}_{31}$, $\hat{b}_{21}$ and $\hat{a}_{21}$ where one addend is a value from a uniform distribution. Therefore, also all the sums are from a uniform distribution for P_1. The sums are independent, because their uniformly distributed components r_{32}, r_{23}, s_{32} and s_{23} have not been addends in messages received before by P_1. □

Theorem 4.5. *Each share in $\{uv_A, uv_B, uv_C\}$ is from a uniform distribution.*

Proof. Each one of the shares uv_A, uv_B and uv_C is a sum in which the addends $(t_{31} - t_{12})$, $(t_{12} - t_{23})$ and $(t_{23} - t_{31})$ are from a uniform distribution. Note, that as shares they represent the value zero which means that locally adding them to another shared value will retain the value while making the shares have a uniform distribution. □

4.5.4 Share conversion from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$

At the start of the protocol each miner has a bit share of a value $[\![u]\!]$ in $\mathbb{Z}_2$. To use such shares in protocols we need to convert them to be uniform in $\mathbb{Z}_{2^{32}}$ without compromising $[\![u]\!]$. This functionality is required to validate the input from the controllers, for example, when range-limited values such as boolean data is received from the controllers. Share conversion is also an important sub-task in the bit extraction protocol described in the next section.

We now explain the design idea behind the protocol. Assume that miners have one-bit shares u_A, u_B and u_C of a bit value $[\![u]\!]$ and we want to compute shares that represent the same bit, but with shares from $\mathbb{Z}_{2^{32}}$. We reduce this task to distributed secure evaluation of the polynomial:

$$f(x_1, x_2, x_3) = x_1 + x_2 + x_3 - 2x_1x_2 - 2x_1x_3 - 2x_2x_3 + 4x_1x_2x_3$$

since we notice that for all bit shares $u_A, u_B, u_C \in \mathbb{Z}_2$ we can compute

$$f(u_A, u_B, u_C) = (u_A + u_B + u_C) \bmod 2$$

over integer values. The protocol is based on evaluating the polynomial f on shares of $[\![u]\!]$ which means that we want to evaluate

$$f(u_A, u_B, u_C) = u_A + u_B + u_C - 2u_Au_B - 2u_Au_C - 2u_Bu_C + 4u_Au_Bu_C \bmod 2^{32} \quad (7)$$

by computing shares of the addends in the sum. The single-value shares u_A, u_B and u_C are available, so we need to compute shares of $[\![u_Au_B]\!]$, $[\![u_Au_C]\!]$, $[\![u_Bu_C]\!]$ and $[\![u_Au_Bu_C]\!]$. We use the same multiplication method by Du and Atallah [DA00] for computing $[\![u_Au_B]\!]$, $[\![u_Au_C]\!]$ and $[\![u_Bu_C]\!]$ as we used in the share multiplication protocol.

Shares of the product $[\![u_Au_Bu_C]\!]$ are computed by running the share multiplication protocol given in Algorithm 10 on shares of $[\![u_Au_B]\!]$ and $[\![u_C]\!]$. For this purpose P_3 additively shares its share u_C with the other miners.

After computing shares of addends in equation (7) we can combine them into shares of $[\![u]\!]$. The complete protocol is described in Algorithm 11. In the first round the parties generate the necessary amount of uniform values and in the second round they run the necessary multiplication sub-protocols. Also in the second round P_3 distributes its share u_C between all three miners.

In the third round the parties complete the computation of two-element products $[\![u_Au_B]\!]$, $[\![u_Au_C]\!]$ and $[\![u_Bu_C]\!]$ and run the share multiplication protocol to find $u_Au_Bu_C$. The protocol is concluded in the fifth round when parties locally evaluate $f(u_A, u_B, u_C)$ with the computed shares and compute shares u'_A, u'_B and u'_C.

Algorithm 11: Share conversion from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$

Data: shared bit $[\![u]\!] = [\![u_A, u_B, u_C]\!]$, $u_A, u_B, u_C \in \mathbb{Z}_2$
Result: shared bit $[\![u']\!] = [\![u'_A, u'_B, u'_C]\!]$, $u'_A, u'_B, u'_C \in \mathbb{Z}_{2^{32}}$

`Round 1`

Alice generates $r_{12}, r_{13}, s_{12}, s_{13}, t_{12} \leftarrow \mathbb{Z}_{2^{32}}$
Alice computes $s_A = r_{12}r_{13} - s_{12} - s_{13}$
Bob generates $r_{23}, r_{21}, s_{23}, s_{21}, t_{23} \leftarrow \mathbb{Z}_{2^{32}}$
Bob computes $s_B = r_{23}r_{21} - s_{23} - s_{21}$
Charlie generates $r_{31}, r_{32}, s_{31}, s_{32}, t_{31} \leftarrow \mathbb{Z}_{2^{32}}$
Charlie computes $s_C = r_{31}r_{32} - s_{31} - s_{32}$
All values $*_{ij}$ are sent from M_i to M_j

`Round 2`

Alice computes $\hat{b}_{12} = r_{31} + u_A$, $\hat{b}_{13} = r_{21} + u_A$
Bob computes $\hat{b}_{23} = r_{12} + u_B$, $\hat{b}_{21} = r_{32} + u_B$
Charlie computes $\hat{b}_{31} = r_{23} + u_C$, $\hat{b}_{32} = r_{13} + u_C$
Additionally, Charlie shares u_C: $c_{31}, c_{32} \leftarrow \mathbb{Z}_{2^{32}}$, $c_{33} = u_C - c_{31} - c_{32}$
All values $*_{ij}$, $i \neq j$ are sent from M_i to M_j

`Round 3`

Alice computes
$ab_A = s_{31} - r_{31}\hat{b}_{21}$
$ac_A = \hat{b}_{31}\hat{b}_{13} + s_{21} - \hat{b}_{31}r_{21}$
$bc_A = s_A$
$c_A = c_{31}$
Bob computes
$ab_B = \hat{b}_{12}\hat{b}_{21} + s_{32} - \hat{b}_{12}r_{32}$
$ac_B = s_B$
$bc_B = s_{12} - r_{12}\hat{b}_{32}$
$c_B = c_{32}$
Charlie computes
$ab_C = s_C$
$ac_C = s_{23} - r_{23}\hat{b}_{13}$
$bc_C = \hat{b}_{23}\hat{b}_{32} + s_{13} - \hat{b}_{23}r_{13}$
$c_C = c_{33}$
The parties run the share multiplication protocol given in Algorithm 10 as a sub-protocol to multiply $[\![ab]\!]$ and $[\![c]\!]$. The result is $[\![abc]\!]$ and the sub-protocol takes two rounds because the local computations can be performed in the current protocol.

`Round 5`

Alice computes $u'_A = u_A - 2ab_A - 2bc_A - 2ac_A + 4abc_A - t_{12} + t_{31}$
Bob computes $u'_B = u_B - 2ab_B - 2bc_B - 2ac_B + 4abc_B - t_{23} + t_{12}$
Charlie computes $u'_C = u_C - 2ab_C - 2bc_C - 2ac_C + 4abc_C - t_{31} + t_{23}$

At the end of the protocol each party has a share of a value $[\![u']\!] = [\![u]\!]$ but with different shares. The conversion requires five rounds and a total of 23 messages plus the cost of a share multiplication operation. Therefore, the total is 50 messages.

Theorem 4.6. *The share conversion protocol is correct.*

Proof. In the first round the parties generate randomness and in the second round they distribute their shares of the bit. In the third round miners start computing the two-element products of the bit shares. We show that the computed shares ab_A, ab_B, ab_C represent the product of u_A and u_B.

$$\begin{aligned} ab_A + ab_B + ab_C &= s_{31} - r_{31}\hat{b}_{21} + \hat{b}_{12}\hat{b}_{21} + s_{32} - \hat{b}_{12}r_{32} + s_C \\ &= s_{31} - r_{31}r_{32} - r_{31}u_B + r_{31}r_{32} + r_{31}u_B \\ &\quad + u_A r_{32} + u_A u_B + s_{32} - r_{31}r_{32} - u_A r_{32} + s_C \\ &= s_{31} + u_A u_B + s_{32} - r_{31}r_{32} + r_{31}r_{32} - s_{31} - s_{32} \\ &= u_A u_B \end{aligned}$$

Similarly it can be shown that $ac_A + ac_B + ac_C = u_A u_C$ and $bc_A + bc_B + bc_C = u_B u_C$. To compute $u_A u_B u_C$, we use the share multiplication protocol to multiply $u_A u_B$ with u_C. For that purpose the third miner shares its bit share u_C beforehand. In the fifth round we have all the necessary shares for evaluating the polynomial f computed so that the miners can evaluate the polynomial with the shares and use the results as shares of $[\![u]\!]$ in $\mathbb{Z}_{2^{32}}$. The addends t_{ij} in the shares are used to give the shares a uniform distribution and they cancel out trivially at reconstruction as $(t_{31} - t_{12}) + (t_{12} - t_{23}) + (t_{23} - t_{31}) = 0$. □

Theorem 4.7. *The share conversion protocol is perfectly secure.*

Proof. To prove the security of the protocol, we show that the messages received by each miner in every round are independent values from a uniform distribution. The protocol is not completely symmetrical as in round 2 the miner P_3 distributes shares of its input to the other miners. If we prove security for P_1 or P_2 then security for P_3 follows directly as it receives less values than the other two nodes.

It is enough to show that messages received by P_1 are independent values from a uniform distribution as the proofs for P_2 and P_3 are very similar. In the first round this is easy to show, as all values received by P_1 are independently generated from a uniform distribution.

In the second round P_1 receives values $\hat{b}_{21}$ and $\hat{b}_{31}$ and c_{31}. The first two are sums of a uniformly distributed value and another value so they too are uniformly distributed and c_{31} is uniformly distributed by construction. Since P_1 has not

received the uniformly distributed operands r_{32} and r_{23} in the sums $\hat{b}_{21}$ and $\hat{b}_{31}$ before, the latter two values are independent. We now need to show that c_{31} is independent from both $\hat{b}_{21}$ and $\hat{b}_{31}$. Since c_{31} is generated from a uniform distribution independently from $\hat{b}_{21}$ and $\hat{b}_{31}$, all three values are independent.

In round 3 the miners run the share multiplication protocol which was proven perfectly secure in Theorem 4.4. □

Theorem 4.8. *Each share in* $\{u'_A, u'_B, u'_C\}$ *is from a uniform distribution.*

Proof. This protocol uses the same technique as the share multiplication protocol in Algorithm 10 to give the output shares a uniform distribution. Namely, each party generates a value t_{ij} that is passed to the next party and is later used in the sum as and addend by one party and as a subtrahend by another. The respective proof is given in Theorem 4.5 and applies also to this protocol. □

4.5.5 Bit extraction

Each node has a share of the value $[\![u]\!]$ and we want to compute shares of the bits that form $[\![u]\!]$. This is a versatile operation that allows us to perform bitwise operations with the shares. For example, in our framework we use bit extraction to build a protocol that evaluates the greater-than predicate. We will now explain our method of securely computing the bits and how it is the structure of the protocol. Note, that in this chapter we use the notation $a(i)$ to denote the i-th bit or bit share of a.

This protocol is inspired by the work of Damgård *et al* [DFK+06]. To compute the bits of a secret shared value $[\![u]\!]$, each party P_j uniformly generates 32 shares $r(0)_j, \ldots, r(31)_j \in \mathbb{Z}_2$ and combines them into a 32-bit share $r_j = \sum_{i=0}^{31} r(i)_i$. Then, the parties compute the difference $[\![a]\!] = [\![u]\!] - [\![r]\!]$. Since the difference of a uniformly distributed value and another value is also from a uniform distribution, each party can publish its share a_j. Now all parties add the two received values with their own one to compute the public value $[\![a]\!]$. Since each party knows the public value a and has shares of the bits of r we have reduced the problem of extracting the bits of a to computing shares of the bits of $a + r$.

$$\begin{array}{cccccc} & c_{31} & \ldots & c_2 & c_1 & \\ & a_{31} & \ldots & a_2 & a_1 & a_0 \\ + & r_{31} & \ldots & r_2 & r_1 & r_0 \\ \hline & u_{31} & \ldots & u_2 & u_1 & u_0 \end{array}$$

Figure 11: Bitwise addition example

Figure 11 shows how the bitwise addition is performed—each party has shares of $[\![a]\!] = [\![u]\!] - [\![r]\!]$ and also shares of bits of $[\![r]\!]$. We use bitwise addition to compute the i-th bit of the sum as follows:

$$\begin{aligned} [\![d(0)]\!] =& [\![a(0)]\!] + [\![r(0)]\!] \\ [\![d(i)]\!] =& 2^i[\![a(i)]\!] + 2^i[\![r(i)]\!] + [\![c(i)]\!], \text{ if } i > 0 \end{aligned} \tag{8}$$

Note that the bits $[\![d(i)]\!]$ are computed in their respective power of two. To compute carry bits, we convert the bit representations of $[\![a]\!]$ and $[\![r]\!]$ into values using the standard base 2 to base 10 conversion procedure. If we have computed n bits $b(i)$ of a value b, we can compute the value itself as follows:

$$b = \sum_{i=0}^{n} 2^i b(i) \ .$$

We compute carry bits for position i by building values

$$\begin{aligned} [\![\hat{a}]\!] &= 2^{i-1}[\![a(i-1)]\!] + \cdots + [\![a(0)]\!] \\ [\![\hat{r}]\!] &= 2^{i-1}[\![r(i-1)]\!] + \cdots + [\![r(0)]\!] \\ [\![\hat{u}]\!] &= 2^{i-1}[\![u(i-1)]\!] + \cdots + [\![u(0)]\!] \end{aligned}$$

and computing $[\![carry]\!] = [\![\hat{a}]\!] + [\![\hat{r}]\!] - [\![\hat{u}]\!]$. The computed carry bit $[\![carry]\!]$ is computed to the i-th power of two which is suitable for use in equation (8).

Unfortunately, the protocol is not as simple as computing the bits $[\![d(i)]\!]$ as we have to prevent the carry bits from leaking. We notice, that while the bit $[\![d(i)]\!]$ correctly represents the desired bit $[\![u(i)]\!]$ as

$$[\![d(i)]\!] = 2^i[\![a(i)]\!] + 2^i[\![r(i)]\!] + [\![carry]\!] = [\![u(i)]\!] \bmod 2,$$

it actually has four possible values

$$[\![d(i)]\!] = 2^i[\![a(i)]\!] + 2^i[\![r(i)]\!] + [\![c(i)]\!] \in \{0, 2^i, 2^{i+1}, 2^{i+1} + 2^i\}.$$

To prevent the extra bit from leaking we proceed by uniformly generating 32 bits $[\![p(0)]\!], \ldots, [\![p(31)]\!]$, one for hiding each bit $[\![d(i)]\!]$, at each party and computing the bits $[\![f_i]\!]$

$$[\![f(i)]\!] = [\![d(i)]\!] + 2^i[\![p(i)]\!] \bmod 2^{i+1} \ .$$

The parties then publish $[\![f_i]\!]$ and evaluate the published value. If the published value is zero, then the i-th bit share of $[\![u]\!]$ is equal to $[\![p(i)]\!]$ and in the opposite case it is $[\![1 - p(i)]\!]$. The rounds of the protocol are given in Algorithm 12.

Algorithm 12: Bit extraction from shares part 1—setup

Data: shared value $[\![u]\!]$

Result: shares of 32 bits $[\![u(0), \ldots, u(31)]\!]$

`Round 1`

Alice generates $r(0)_A, \ldots, r(31)_A, p(0)_A, \ldots, p(31)_A \leftarrow \mathbb{Z}_2$

Alice computes $q(i)_A = (e + 1) \bmod 2$, $i = 0, \ldots, 31$.

Bob generates $r(0)_B, \ldots, r(31)_B, p(0)_B, \ldots, p(31)_B \leftarrow \mathbb{Z}_2$

Bob computes $q(i)_B = (e_B + 1) \bmod 2$, $i = 0, \ldots, 31$.

Charlie generates $r(0)_C, \ldots, r(31)_C, p(0)_C, \ldots, p(31)_C \leftarrow \mathbb{Z}_2$

Charlie computes $q(i)_C = (e_C + 1) \bmod 2$, $i = 0, \ldots, 31$.

The parties run the share conversion protocol given in Algorithm 11 to convert the bits $[\![r(i)]\!]$, $[\![p(i)]\!]$ and $[\![q(i)]\!]$ to shares in $\mathbb{Z}_{2^{32}}$. The computation requires four rounds as the local computations are performed within the current protocol.

`Round 5`

Alice computes

$r_A = \sum_{j=0}^{31} 2^j \hat{r}(j)_A$

$v_{11} = u_A - r_A$

Bob computes

$r_B = \sum_{j=0}^{31} 2^j \hat{r}(j)_B$

$v_{21} = u_B - r_B$

Charlie computes

$r_C = \sum_{j=0}^{31} 2^j \hat{r}(j)_C$

$v_{31} = u_C - r_C$

All values $*_{ij}$, $i \neq j$ are sent from M_i to M_j

`Round 6`

Alice:

computes $a_A = v_{11} + v_{21} + v_{31}$ and divides it into bits $a(0)_A, \ldots, a(31)_A$

$d(0)_A = a(0)_A + r(0)_A$

$f(0)_A = d(0)_A + p(0)_A \bmod 2^{i+1}$

sends $f(0)_A$ to Bob and Charlie

Bob:

initialises $a_B = 0$ and divides it into bits $a(0)_B, \ldots, a(31)_B$

$d(0)_B = a(0)_B + r(0)_B$

$f(0)_B = d(0)_B + p(0)_B \bmod 2^{i+1}$

sends $f(0)_B$ to Bob and Charlie

Charlie:

initialises $a_C = 0$ and divides it into bits $a(0)_C, \ldots, a(31)_C$

$d(0)_C = a(0)_C + r(0)_C$

$f(0)_C = d(0)_C + p(0)_C \bmod 2^{i+1}$

sends $f(0)_C$ to Bob and Charlie

for $i = 1$ **to** 31 **do**

`Round 6 + i`

Alice:

$f(i-1) = f(i-1)_A + f(i-1)_B + f(i-1)_C$

if $f(i-1) = 2^i$ **then** $u(i-1)_A = q(i-1)_A$

else $u(i-1)_A = p(i-1)_A$

$\hat{a}_A = \sum_{j=0}^{i-1} a(j)_A$

$\hat{r}_A = \sum_{j=0}^{i-1} r(j)_A$

$\hat{u}_A = \sum_{j=0}^{i-1} u(j)_A$

$carry_A = \hat{a}_A + \hat{r}_A - \hat{u}_A$

$d(i)_A = a(i)_A + 2^i r(i)_A + carry_A$

$f(i)_A = d(i)_A + 2^i p(i)_A \bmod 2^{i+1}$

sends $f(i)_A$ to Bob and Charlie

Bob:

$f(i-1) = f(i-1)_A + f(i-1)_B + f(i-1)_C$

if $f(i-1) = 2^i$ **then** $u(i-1)_B = q(i-1)_B$

else $u(i-1)_B = p(i-1)_B$

$\hat{a}_B = \sum_{j=0}^{i-1} a(j)_B$

$\hat{r}_B = \sum_{j=0}^{i-1} r(j)_B$

$\hat{u}_B = \sum_{j=0}^{i-1} u(j)_B$

$carry_B = \hat{a}_B + \hat{r}_B - \hat{u}_B$

$d(i)_B = a(i)_B + 2^i r(i)_B + carry_B$

$f(i)_B = d(i)_B + 2^i p(i)_B \bmod 2^{i+1}$

sends $f(i)_B$ to Bob and Charlie

Charlie:

$f(i-1) = f(i-1)_A + f(i-1)_B + f(i-1)_C$

if $f(i-1) = 2^i$ **then** $u(i-1)_B = q(i-1)_B$

else $u(i-1)_B = p(i-1)_B$

$\hat{a}_B = \sum_{j=0}^{i-1} a(j)_B$

$\hat{r}_B = \sum_{j=0}^{i-1} r(j)_B$

$\hat{u}_B = \sum_{j=0}^{i-1} u(j)_B$

$carry_B = \hat{a}_B + \hat{r}_B - \hat{u}_B$

$d(i)_B = a(i)_B + 2^i r(i)_B + carry_B$

$f(i)_B = d(i)_B + 2^i p(i)_B \bmod 2^{i+1}$

sends $f(i)_B$ to Bob and Charlie

Round 38

Alice:

$f(31) = f(31)_A + f(31)_B + f(31)_C$

if $f(31) = 2^{31}$ **then** $u(31)_A = q(31)_A$

else $u(31)_A = p(31)_A$

Bob:

$f(31) = f(31)_A + f(31)_B + f(31)_C$

if $f(31) = 2^{31}$ **then** $u(31)_B = q(31)_B$

else $u(31)_B = p(31)_B$

Charlie:

$f(31) = f(31)_A + f(31)_B + f(31)_C$

if $f(31) = 2^{31}$ **then** $u(31)_B = q(31)_B$

else $u(31)_B = p(31)_B$

After running the protocol, each node has shares of bits $[\![u(0), \ldots, u(31)]\!]$ which represent the bits of the input value $[\![u]\!]$. In the first round we run 96 share conversions, but since we can vectorise this operation, we can do all the conversions with the same number of messages as a single conversion. Each message will contain data elements for all parallel operations. This means that the first round requires 50 messages. Two messages are sent in the fifth round and three messages are sent each round from sixth to 37th. This gives us a total of 38 rounds and $50 + 2 + 32 \times 6 = 244$ messages.

Theorem 4.9. *The bit extraction protocol is correct.*

Proof. To show correctness we divide this protocol into subprotocols and prove that they perform the desired tasks and also jointly form a correct protocol. In rounds 1 and 5 we generate uniformly distributed values for the protocol and compute the difference $[\![a]\!] = [\![u]\!] - [\![r]\!]$. The correctness of this step is trivial.

Now we have reduced the task to finding shares of bits $[\![u(i)]\!]$ of the sum $[\![a]\!]+[\![r]\!]$ where each party has shares of the difference $[\![a]\!]$ and the bits of $[\![r]\!]$. To prove correctness, we describe the computation path of bits $[\![u(i)]\!]$. Shares $[\![u(i)]\!]$ are computed based on bits $f(i)$ so we will show how they are computed. At first, bits $[\![d(i)]\!]$ are computed by adding the respective bits of $[\![a]\!]$ and $[\![r]\!]$ and the carry bit if available.

$$[\![d(i)]\!] = 2^i[\![a(i)]\!] + 2^i[\![r(i)]\!] + [\![c(i)]\!] \tag{9}$$

In equation (9) $[\![c(i)]\!]$ is the respective carry bit in the required power of two when $i > 0$ or 0, if $i = 0$. The bits $[\![d(i)]\!]$ are candidates for being a bit of $[\![u(i)]\!]$ and we

verify them by using the following procedure. We add a uniformly generated bit $[\![p(i)]\!]$ modulo 2^{i+1} to strip the carry bit from the sum and stop it from leaking:

$$[\![f(i)]\!] = [\![d(i)]\!] + [\![p(i)]\!] \bmod 2^{i+1} \ .$$

Then, we publish the bit $[\![f(i)]\!]$ between all nodes so that each party learns the sum $f(i) + p(i)$. Now each party determines the bit share $[\![u(i)]\!]$ by checking the i-th bit of $f(i)$ and comparing it to 1. If the bit is 0 we know that $[\![d(i)]\!] = [\![p(i)]\!]$ because $0 + 0 = 0 \bmod 2$ and $1 + 1 = 0 \bmod 2$ or more simply—the respective bits of $[\![d(i)]\!]$ and $[\![p(i)]\!]$ must be equal for their sum to be 0 modulo 2. In this case $[\![u(i)]\!] = [\![p(i)]\!]$.

In the opposite case, if the i-th bit of $f[i]$ is 1 then we know that the respective bits of $[\![d(i)]\!]$ and $[\![p(i)]\!]$ must be different. For that reason exactly we generate bits $[\![q(i)]\!] = 1 - [\![p(i)]\!]$ in the beginning of the protocol—we can now say that $[\![u(i)]\!] = [\![q(i)]\!]$.

The only thing left to show for correctness is carry bit computation. To compute the carry bit for $[\![d(i)]\!], i > 0$, we combine bits $0, \ldots, i-1$ of $[\![a]\!]$, $[\![r]\!]$ and $[\![u]\!]$ into values $[\![\hat{a}]\!]$, $[\![\hat{r}]\!]$ and $[\![\hat{u}]\!]$. Then, each node computes

$$carry = [\![a]\!] + [\![r]\!] - [\![u]\!]$$

and by that, removes all bits except for the carry from the sum so that the carry bit remains in the i-th position of *carry*. □

Theorem 4.10. *The bit extraction protocol is perfectly secure.*

Proof. We will now present a sketch for the proof that the protocol is perfectly secure. To show that the protocol is secure we show that it is composed of secure sub-protocols. In the first round, parties run share conversion on a vector of 96 bit shares. This sub-protocol was proved perfectly secure in Theorem 4.7.

In the fifth round, P_1 receives values v_{21} and v_{31} which are differences between the input shares u_* and a generated uniformly distributed value r_*. The values r_* are compiled by the sending parties by combining shares of 32 bits generated independently from a uniform distribution so the sums v_{21} and v_{31} are also uniformly distributed independent values.

The other rounds share the message structure so the following discussion applies to rounds $4, \ldots, 35$. The messages received by P_1 are in the form $f(i)_* = d(i)_* + 2^i p(i)_*$. In these messages, $p(i)_*$ is a uniformly distributed value so the sums are also uniformly distributed. Since the values $p[i]_*$ have not been used in any other protocol messages they are also independent for P_1.

From the security of the sub-protocols we conclude that the bit extraction protocol is perfectly secure. □

Theorem 4.11. *Each share in* $\{u(i)_j : i \in \{0, \ldots, 31\}, j \in \{A, B, C\}$ *is from a uniform distribution.*

Proof. Each share $u(i)_j$ is computed from either $p(i)_j$ or $q(i)_j = 1 - p(i)_j$ which are independently and uniformly generated bit shares converted to be a shares in $\mathbb{Z}_{2^{32}}$ in the first round. Since we proved that the shares of converted bits are from a uniform distribution in Theorem 4.8 it follows, that also shares $u(i)_j$ are uniformly distributed. □

4.5.6 Evaluating the greater-than predicate

All the protocols we have described up to now have performed mathematical operations with shared values with outputs also being a shared value. We now describe a protocol that evaluates a predicate with two parameters—the greater-than comparison. It is designed as an example of how to build predicate operations in our framework.

Since $\mathbb{Z}_{2^{32}}$ is a finite ring we have to define the greater-than predicate for its elements. We must take into account that all arithmetic is performed modulo 2^{32} and elements of $\mathbb{Z}_{2^{32}}$ natively have no sign. We define our predicate $\mathsf{GT} : \mathbb{Z}_{2^{32}} \times \mathbb{Z}_{2^{32}} \to 0, 1$ as follows:

$$\mathsf{GT}(x, y) = \begin{cases} 1, & \text{if the last bit of the difference } x - y \text{ is } 1 \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

This definition is correct for 31-bit integers and fails for 32-bit integers as the latter requires the operation to return a value in $\mathbb{Z}_{2^{33}}$. However, 31-bit integers are sufficient for practical purposes. The reason for failure is that the last bit of the value becomes the sign bit which basically divides the set of values into two parts—the "positive" half which consists of values $0, \ldots, 2^{30} - 1$ and the "negative" half which contains $2^{30}, \ldots, 2^{31} - 1$. This is similar to the way programming languages handle signed integers. It is important to remember this while using the framework, because for any value $z \in \{2^{30}, \ldots, 2^{31} - 1\}$ the predicate $\mathsf{GT}(z, 1)$ is false as the value z is interpreted as a "negative" value.

Algorithm 15 shows the protocol for evaluating GT on two additively shared values.

Algorithm 13: Evaluating the greater-than predicate

Data: shared values $[\![u]\!]$ and $[\![v]\!]$
Result: $f = \mathsf{GT}(u, v)$ according to the definition in equation (10)
`Round 1`
Alice computes $d_A = u_A - v_A$.
Bob computes $d_B = u_B - v_B$.
Charlie computes $d_C = u_C - v_C$.
The parties run the bit extraction protocol given in Algorithm 12to find the bits $[\![d(0)]\!], \ldots, [\![d(31)]\!]$ of $[\![d]\!]$. The computation requires 37 rounds as the local computations are performed within the current protocol.
`Round 38`
Alice learns the result share $[\![d(31)_A]\!]$.
Bob learns the result share $[\![d(31)_B]\!]$.
Charlie learns the result share $[\![d(31)_C]\!]$.

At the end of the protocol the miners have shares of the bit $d(31)_A$, $d(31)_B$ and $d(31)_C$. Based on this bit the miners may now push the share of the greater value on the stack top or report the value of the predicate to the controller. Both approaches are viable, but the first one is preferred, because it does not reveal any information. Revealing the bit allows the controller to change its control flow dynamically during the algorithm, but has the obvious downside of leaking a bit of information. The protocol requires as many messages as are needed for bit extraction plus the three shares sent to the controller which gives us a grand total of $244 + 3 = 247$ messages.

Theorem 4.12. *The greater-than protocol is both correct and perfectly secure.*

Proof. The correctness of the protocol follows from the facts that it corresponds to our definition and GT and the bit extraction protocol was proven correct in Theorem 4.9.

The security is trivially reduced to the one of bit extraction that was also proved in Theorem 4.10. □

Theorem 4.13. *Each share in* $\{d(31)_A, d(31)_B, d(31)_C\}$ *is from a uniform distribution.*

Proof. Since the shares are bits from the bit extraction algorithm presented in Algorithm 12 the uniformity of their distribution follows directly from Theorem 4.11. □

5 Overview of our implementation

5.1 Overview

Based on the described secure computation framework, we have built an implementation called Sharemind. The name reflects the functionality—data mining of secret shared databases. There is also a more metaphorical explanation—it can be imagined that the three servers performing computations form a shared mind.

Our implementation is in the form of a software program for personal computers. There are two kinds of programs—mining nodes run the miner software and other parties run the controller software. The miner software consists of algorithms and protocols that perform the multiparty computation and data mining tasks. The controller software is used to send data and commands to the miner software.

All the protocols described in the framework are implemented in the miner software. The controller software orders the miners to execute the protocols in the desired order. Since we need controllers with different functionalities, like data entry terminals and analysis workstations, we have developed a controller library which makes developing such applications straightforward. The programmer will use the methods in the library to connect to the miners, share data and execute protocols.

This section gives an overview of the parties' communication channel set-up process. We also describe the computational capabilities of the miners and the controller library interface. However, we start by giving the reader some technical information about the implementation.

5.2 Implementation notes

The software is implemented in the C++ programming language using the GNU C++ compiler. It uses the RakNet network library for communication[1]. The system is designed to be cross-platform and development is done on Apple Mac OS X, Linux/UNIX and Microsoft Windows. We chose C++ and RakNet because the network library is designed for high throughput and it also works with popular computing platforms. A possible downside is that all parties in the system must communicate by using RakNet, but this is a low risk, because the library has an open source license.

Operations in $\mathbb{Z}_{2^{32}}$ are provided by the C++ native 32-bit unsigned long datatype. The messages are sent over the network by using the UDP transport protocol which is one of the most common Internet protocols. UDP datagrams have less overhead and are smaller than TCP packets. However, plain UDP is unreliable so we use the reliability layer offered by the RakNet network library. We

[1] RakNet — a reliable cross-platform network library. URL: http://www.rakkarsoft.com/

prefer using RakNet over TCP because of smaller packages, easier programming interface and platform independence.

There is an important restriction in the current version of Sharemind at the time of defending this thesis—only one controller party is supported by the miners at any time. In a future version we might extend the scheduler to support more controller nodes. The scheduler can also be improved to minimise the amount of wasted computing cycles during network transfers. We also evaluate the risks of executing multiple parallel queries on the miners as the number of values available for our honest-but-curious miners will grow with each such query.

The current implementation has no restrictions on data retrieval—in the presented implementation the miners happily provide shares of all data in their stacks, heaps and databases. This is for testing purposes and helps to verify the results of computation. In a real-world scenario the miners must withhold all raw data. As a final note, the communication in the current version of Sharemind is unencrypted.

5.3 Communication channels

In our system all miners have communication channels with each other miner and the controller has a communication line to each miner. During controller setup the library locates all three miners and assigns node numbers 1, 2 and 3 to them. After that the miners connect to each other and at the end of start-up all nodes have communication channels to other nodes in the system.

Although UDP is a connectionless protocol, a persistent connection is maintained for each channel by the network library. This allows us to detect transmission problems and node exits. The communication between nodes is message-based with ordering and reliability provided by the library. There are multiple ways of sending messages—using predefined fixed structures or dynamically assembled bitstreams. In the first case the messages always have a fixed size and structure but in the second case we have exact control of the amount of data in the message. For example, we can put the number of transmitted values in the beginning of the message and follow it by all the values. We chose the dynamic construction option because it allows us to easily transfer vectors of any size between parties.

5.3.1 Messaging module

Both the miners and controllers use a common messaging class which is called the `NetworkNode`. Each party runs one `NetworkNode` instance that takes care of network setup and messaging. Messages are received and cached by the `NetworkNode` so when the miner or controller expects a value from another node it requests this value from the network node. If the `NetworkNode` has the requested value in

its queues, it returns it immediately. Otherwise it waits until the desired value arrives.

The communication is composed of three types of values—elements of $\mathbb{Z}_{2^{32}}$, vectors of those elements and strings. The first two are used in passing command codes and parameters and strings are currently used only to specify database names in loading and saving commands.

5.3.2 Prototyping and real world application

The communication infrastructure in Sharemind is designed from a prototyping perspective and a real-world system will have to be built on slightly different principles. Whereas in Sharemind the set-up procedure is coordinated by the controller to simplify testing new applications, in a real-world solution the miners must be set up individually. Sharemind is a self-organising system and this makes developing easier, because running an application requires less manual configuration.

We now describe the preferred scenario for an application with higher security requirements. Three organisations should be identified to maintain the servers that run the miners. These organisations must have no motive to collaborate with each other to reconstruct the database. This can be achieved if the organisations are competing companies or government organisations protecting people's privacy. Each miner maintainer will ensure that the miner runs and is available for the controller applications. The controller applications are configured with the network addresses of the miners and connect to them at startup. The main difference with Sharemind is that in our implementation miners start with no configuration—it is supplied by the controller when it starts. In the real world the miners are configured locally and the controllers have no control over their operation.

In Sharemind the controllers can request any data in the stack, heap or database from the miners. This allows us to use automated tests to verify the behaviour of the system. In a real world database the miners will not give out shares of private information.

5.4 Computational environment

5.4.1 Overview

If we want to run data mining algorithms on the system, we need to provide a computation environment for them to run in. To implement an algorithm we require a processor that can run instructions and storage space for keeping intermediate results. Additionally, permanent storage allows the algorithms to save time on loading all the data from the controller before each computation.

In our implementation the three miners form a distributed virtual processor for

secure multiparty computations. This processor has an instruction scheduler for running operations, a stack for passing arguments for operations, a heap for addressable storage of intermediate results and a database for long-term data storage. The processor also has communication ports for controllers that can load instructions and data to the processor for processing.

Note that all miners run the same program code. If some query requires specific processing meaning that some nodes must perform differently as others, it is achieved by the miner reading its assigned number and performing accordingly. The miners are numbered from one to three. Another consequence is that the protocols are rewritten to address other nodes by relative position instead of the absolute one. For example, instead of receiving some data from miner 1, processing it and forwarding it to miner 3, the code receives data from the previous miner, processes it and sends it to the next miner. The previous and next relationship are defined cyclically over the numbers of miners, with the wrapping point between miners three and one.

5.4.2 Instruction scheduler

Each miner has a simple instruction scheduler that handles incoming commands and determines the order of processing. The miners process instructions sequentially, one operation at a time. The order is determined by the arrival of messages containing the commands. All miners run the same operation in parallel.

Each scheduler manages a queue of instructions. Single round operations, like data manipulation and simple arithmetic are executed immediately without using the instruction queue. In the context of our system, operations that have more than one round are called *queries*. Queries are added to the queue and the scheduler processes them round by round. Algorithm 14 illustrates how the scheduler decides the execution order of operations.

Algorithm 14: SHAREMIND instruction scheduler

```
Data: list of queries Q
while the scheduler is running do
    if Q contains a query then
        let q be the first query in Q
        if q is not complete then
            complete next round of q
        else
            remove q from Q
        end
    else
        let c be the next command
        if c is a single-round instruction then
            process c
        else
            add c to Q
        end
    end
end
```

After each round the query passes control back to the scheduler that checks if the query is complete and if it is, discards it. In either case it takes the first query in the queue and runs its next round. The processing is sequential—no new operations are executed and no new queries are added until the queue is empty.

Both the controllers and the miners themselves can add operations to the queue. For example, a bit extraction protocol requests the scheduler to add a share conversion query to the front of the queue. Now if the bit extraction query completes its round, the share conversion will be run next. If the share conversion completes, the scheduler will run the next round of the bit extraction query.

It is possible to improve the scheduling of operations to reduce or even completely remove the idle time of the miners as they wait for network messages. This can be achieved by switching between multiple queries as they wait for data from the other parties.

5.4.3 Runtime storage

The virtual processor has two kinds of runtime storage—the stack and the heap. Their contents are temporary, as they are forgotten when the miner is shut down. Both containers only hold shares of data instead of the actual values. When a value is stored in either one by a controller, the value is shared automatically. Similarly, if a value is read from either the stack of the heap, its shares are read

from the miners and the original value is reconstructed.

The stack represents the well-known data structure with the same name. It provides the standard pushing, peeking and popping operations. The stack is used to pass parameters between processor instructions. Each operations reads its input from the stack and writes the result back on top of it.

The stack has a unified data type—it stores elements of $\mathbb{Z}_{2^{32}}$. Our system also handles vectors of these elements, so the stack must also support pushing and popping vectors. Our solution is to extend our implementation of the stack to a randomly accesible array. This allows us to optimally implement the vector pop operation and retain the order of the elements. We can return the necessary part of the vector without popping each value individually and building a vector from them.

To illustrate the use of the stack for passing arguments to operations, we describe the invocation of the bit extraction query. We start by putting the input value on the stack. Note that in this example we assume that the controller can ask the miners to provide shares of the data at any time. This is for demonstrational purposes only, as a normal system will not publish secret values on command.

1. The controller library receives a command to push a value on the stack and the value is included as a parameter.
2. The controller library distributes the input value into shares.
3. The controller library sends each miner the command to push a value on the stack. Each miner gets a different share of the input value.
4. The miners receive the share and push it on their local stacks.
5. The miners report that the value is on the stack.
6. The controller library receives the report that the value is on the stack.

Assuming that the controller is the initiating party of the query, the following steps are taken to complete the computation.

1. The controller library receives a command to extract bits from a value.
2. The controller library sends each miner the command to extract bits from a value.
3. The miners receive the command and check the state of the stack. If there are no values, the miners report this to the controller as an error and stop.

4. The miners execute the bit extraction query. This query pops a value of the stack and uses it as the input value.

5. The miners complete the query and push 32 shares of bits of the input value on the stack.

6. The miners report to the controller that the query was completed.

The controller may now request that the 32-bit vector of values is popped from the stack and sent to the controller that can then reconstruct the bits from the shares.

The heap is implemented as a randomly accessible vector of elements of $\mathbb{Z}_{2^{32}}$. It is used for storing intermediate results in algorithms. Since elements of this structure can be addressed by indices, the programmer can "put away" results and return to them later. Values in the heap may come from the stack and also directly from the controller.

All data cells in the heap also have one unified type. The heap has no explicit support for vectors and it is up to the programmer to store values in a vector to individual heap cells. Data on the heap can be pushed back on the stack for further operations.

We note that queries also have internal storage for storing protocol data between rounds. This data is kept separately from the stack and the heap. If the query supports vector processing it can also store vectors of input, intermediate and output data. This data cannot be extracted from the query neither to the stack nor the heap.

5.4.4 Long-term storage

To securely store data for further analysis the miners have a secret shared database. The database is in the form of a matrix of elements of $\mathbb{Z}_{2^{32}}$. Following the traditions of runtime storage only shares are stored in the database. The database can be both read and written many times as it can be saved to a disk. The security of the data on the disk is provided by the fact that it is secret shared and will provide no information to the reader without the other shares.

The miners can handle one database at a time, but they are capable of loading and saving databases with different names. Values in the database can be pushed on the stack individually, by column and by row. Values are addressed by a combination of row and column indices. Rows and columns are addressed by their respective indices. There is no diret way of storing a database value in the heap because the heap is designed for intermediate results. Input data for computations is read from the stack in all cases. To save data in the database it must be read either from the stack or the heap and stored in a database cell.

5.5 Controller library

The controller library is the programming API for developing secure computation software on the virtual processor of SHAREMIND. It helps the programmer to correctly communicate with the miners, handle data and run queries. It takes care of connecting to the miners and setting up the computation environment. Methods of the library perform secret sharing and reconstruction where necessary.

Only individual atomic operations are provided by the library. The control structures are written in C++ by the developer of the application. A secure computation program looks like a standard C++ application that uses a specific library to perform computations. The library provides the guarantee that all computations performed by using it preserve the privacy of the data. It is important to understand that all computation performed on the data with other methods than the ones in the controller library are inherently insecure.

For each command given to it, the library performs a remote procedure call to all three miners. The code for the command is sent to the miners, followed by the given parameters. Single values, vectors and strings are serialised before network transmission and recovered at the receiver. Both the miners and the controller use the same network transport layer to exchange commands and data.

Method invocation is blocking—the controller library waits until the miners have completed the task and reported back to the controller with their results. These results are interpreted and the result is given to the programmer using the library method return value. If the results contain data, it is reconstructed from shares automatically,

The library also provides other services like logging, database management and random value generation. These are used by the methods of the library, but they are also available for the programmer.

5.5.1 Controller interface

The developer has access to the miners' functionality through the controller interface class. For the developer the interface looks like a processor with a limited set of commands. The fact that behind the scenes, secret sharing, remote procedure calls and share reconstruction are transparently performed, is hidden from the developer.

Methods of the controller library are divided into the following categories:

1. system operations — setting up the system and handling the database,
2. data management — exchanging data between the controller and the miners, local operations at the miner,
3. computation — performing computations with the data.

For a complete description of the controller interface please see the Controller API documentation of SHAREMIND provided with the source code [Bog07]. Here we give a short overview of the capabilities of the system.

The system operations handle miner configuration and starting connections. The programmer can specify the locations of three miners and order the controller interface to perform the self-organising startup routine that interconnects the miners and prepares the system for computation. Other system operations order the miners to load or save a database with the given name.

Data transfer operations include reading and storing values in the heap by specifying the cell address. The controller can also perform standard stack operations like pushing and popping values in the stack. While these operations exchange data between the miners and the controller there are also local data management operations that are executed at the miners without data transmissions. These include duplicating and swapping values on the stack, pushing contents of the database onto the stack and moving data between the stack and the heap.

Most of the operations work on vectors with multiple elements for performance reasons. This helps us to keep the number of network messages low and our development has confirmed that this is an important optimisation. Sending many small messages is inherently slower than sending one or more larger messages.

Computation operations can be divided into in two categories—single round protocols and queries which have more than one round. Operations which can be completed locally include addition and multiplication by scalar. Currently multi-round operations are multiplication, share conversion from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$, bit extraction and evaluating the greater-than predicate. Note, that most of the computation operations are also vectorised for performance reasons.

5.5.2 Structure of a secure computation application

A secure computation application built to work with SHAREMIND is a regular C++ program that links against the controller library. At startup the application creates a controller interface object that will be used for performing tasks on SHAREMIND miners. The application must specify the location of the miners and initiate the startup routine which interconnects the miners. This routine is blocking and returns when the system is ready for computation.

Now the application can start executing operations on the miners, but for this we need some data. We can provide it directly by sending it to the miners or by loading a previously prepared database. Before each computation we must make sure that there are enough values on the stack and after each computation we must do something with results. If we want to continue computation on the results then we can keep them on the stack and run the next operation.

The controller application orders computation by calling the respective method

of the controller library. The method returns when the computation is done so we can start another one. When we are done, we tell the controller to shut down and the connection to the miners is closed.

5.6 Other implementations

This section gives an overview of other efforts in implementing platforms for secure multiparty computations. We introduce two related projects —Fairplay and SIMAP.

The first implementation we describe is FairPlay [MNPS04] by Malkhi *et al.* FairPlay is a two-party computation system for secure function evaluation. FairPlay specifies computed functions using a secure function definition language SFDL and compiles them to Boolean circuits specified in the secure hardware definition language SHDL. The circuits can then be evaluated on two interpreters—one for each party in the system. The system provides security in the presence of a malicious adversary with a marginal error rate.

SIMAP — Secure Information Management and Processing is a project lead by Peter Bogetoft and Ivan Damgård. The goal of SIMAP is to build a development platform for privacy-preserving data processing applications. The system has evolved from work presented in [BDJ+06] and is currently in development. The computations are executed by using a secure multiparty computation runtime SMCR which consists of two kinds of parties—clients that provide the inputs and servers that perform the computations. A homomorphic threshold secret sharing scheme is used to distribute data between the servers and the runtime is capable of performing operations with this data. The authors have developed a domain-specific language for describing secure computations [NS07].

Our implementation concentrates on building a privacy-preserving interpreter for data analysis algorithms. The interface of the interpreter is currently in the form of a programming library. It would be interesting to investigate the possibility of adapting the high-level languages developed for FairPlay and SIMAP to our interpreter.

6 Experiment results

6.1 Experiment set-up

To evaluate our implementation we benchmarked the computation operations implemented in SHAREMIND. We chose a number of operations and ran them on the system while measuring the time it took to complete each operation. The timing was performed at the controller application which means that the time used to send the command and parameters and receive the results is included in the timings.

We used generated data for input and executed each operation 10 000 times. The operations were timed by using the millisecond-precision timer library provided with the RakNet networking library. After the completion of each step the current timestamp was stored and in the end of the operation the timings were written to a file on the disk.

The testing computers have 2.4 GHz Intel Pentium processors with Hyper-Threading support and 512 megabytes of RAM and run SuSE Linux 10.0 as the operating system. The computers are connected in a local switched network with transmission speeds up to 1 gigabyte per second. The average ICMP ping time for 100 packets exchanged between two computers was 0.191 milliseconds.

In the following all timings are given in milliseconds unless specified otherwise.

6.2 Computing the scalar product

We test multiplication and addition by computing the scalar product of two vectors. This allows us to use the vectorised multiplication capability of SHAREMIND. The timed operation consists of the following steps:

1. Load a database vector with n elements and push it on the stack
2. Load another database vector with n elements and push it on the stack
3. Perform share multiplication on the two loaded n-element vectors.
4. Add the top n values on the stack.
5. Return the result to the controller.
6. Clear the stack for the next iteration.

We ran the operation with several databases with n rows and 10 columns where $n = 5, 100, 1000, 5000, 10000, 20000, 40000, 60000, 80000, 100000$ to see how well the vectorised multiplication operation scales when it has to work with larger

input vectors. In each iteration we randomly picked two database columns and computed their scalar product.

The resulting datasets contain method timings for each iteration of the scalar product algorithm. Based on this data we computed the minimum, maximum, mean, median and standard deviation for the execution time of each method. The minimum execution gives us a lower bound for computation while the maximum execution time represents the possible worst case for a vector of the given size. The mean gives us the average of the computation times in all iterations and the median shows the middle value in the distribution. The median is not affected by extremely small or large values in the distribution. The standard deviation shows how widely the values are distributed. We chose standard deviation, because we are interested in the deviations in each experiment. Standard deviation is also not significantly affected by the few extreme values in our experiment data.

The results for each vector size are given in Tables 1-10. Table 11 gives a comparison of the total computation times for all vector sizes.

The unusually large maximal values occurred in the first iteration of tests with vectors with more than one thousand elements. Starting from the second or third iteration the timings approached the median and varied considerably less. Since we loaded the database before starting timing the computations, the cause for the long first iteration is probably in the operating system or network library level.

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	35	76	48.7	48	3.02
Load 2	36	64	48.59	48	2.65
Multiply	84	121	101.1	96	6.41
Add	36	65	48.61	48	2.7
Read	35	64	48.57	48	2.65
Clear	36	64	48.52	48	2.57
Total	311	421	344.09	337	14.38

Table 1: Scalar product of 5-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	36	120	48.76	48	3.1
Load 2	36	84	48.56	48	2.65
Multiply	83	479	99.58	96	6.95
Add	35	68	48.57	48	2.68
Read	36	64	48.53	48	2.54
Clear	36	65	48.44	48	2.37
Total	263	719	342.43	337	14.53

Table 2: Scalar product of 100-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	35	64	48.75	48	3.27
Load 2	35	65	48.56	48	3.02
Multiply	96	1213	247.82	252	14.53
Add	35	64	48.65	48	3.02
Read	35	72	48.6	48	2.95
Clear	35	64	48.57	48	2.93
Total	336	1453	491.04	492	9.63

Table 3: Scalar product of 1000-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	35	72	48.78	48	3.33
Load 2	35	64	48.61	48	3.03
Multiply	312	6792	692.25	672	155.85
Add	35	65	48.61	48	3.07
Read	35	64	48.52	48	2.95
Clear	35	72	48.43	48	2.92
Total	564	7033	935.19	912	155.75

Table 4: Scalar product of 5 000-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	39	72	49.41	48	3.87
Load 2	40	64	49.22	48	3.48
Multiply	360	4428	1385.69	708	145.96
Add	36	65	49.09	48	3.44
Read	40	65	48.96	48	3.24
Clear	40	64	48.93	48	3.23
Total	600	7188	973.34	949	145.93

Table 5: Scalar product of 10 000-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	39	71	49.52	48	4
Load 2	40	64	49.17	48	3.54
Multiply	456	13116	1014.65	1008	196.39
Add	40	65	49.21	48	3.6
Read	39	68	49.05	48	3.38
Clear	40	64	48.98	48	3.28
Total	720	13356	1260.58	1248	196.17

Table 6: Scalar product of 20 000-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	39	72	49.57	48	4.06
Load 2	40	64	49.23	48	3.62
Multiply	539	6037	1323.84	1285	225.06
Add	39	72	49.26	48	3.67
Read	40	64	49.14	48	3.49
Clear	44	64	49.02	48	3.34
Total	779	6290	1570.07	1536	225.22

Table 7: Scalar product of 40 000-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	38	65	49.5	48	3.96
Load 2	40	64	49.2	48	3.59
Multiply	804	6323	1372.39	1321	252.56
Add	40	68	49.24	48	3.64
Read	39	64	49.11	48	3.47
Clear	46	64	49.03	48	3.35
Total	1081	6575	1618.47	1571	252.52

Table 8: Scalar product of 60 000-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	36	72	49.41	48	3.88
Load 2	40	68	49.22	48	3.61
Multiply	769	6888	1385.69	1344	253.02
Add	40	65	49.17	48	3.55
Read	40	65	49.03	48	3.34
Clear	36	64	48.91	48	3.18
Total	1008	7188	1631.44	1588	253.12

Table 9: Scalar product of 80 000-element vectors, 10 000 iterations

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load 1	39	65	49.63	48	4.1
Load 2	40	68	49.28	48	3.7
Multiply	889	6900	1383.38	1332	238.06
Add	40	64	49.31	48	3.72
Read	40	72	49.14	48	3.51
Clear	40	64	49.02	48	3.35
Total	1139	7187	1629.76	1573	238.1

Table 10: Scalar product of 100 000-element vectors, 10 000 iterations

Size of vector	Minimum	Maximum	Mean	Median	Standard deviation
5	311	421	344.09	337	14.38
100	263	719	342.43	337	14.53
1000	336	1453	491.04	492	9.63
5 000	564	7033	935.19	912	155.75
10 000	600	4668	973.34	949	145.93
20 000	720	13356	1260.58	1248	196.17
40 000	779	6290	1570.07	1536	225.22
60 000	1081	6575	1618.47	1571	252.52
80 000	1008	7188	1631.44	1588	253.12
100 000	1139	7187	1629.76	1573	238.1

Table 11: Scalability of scalar product computations using total computation time in milliseconds

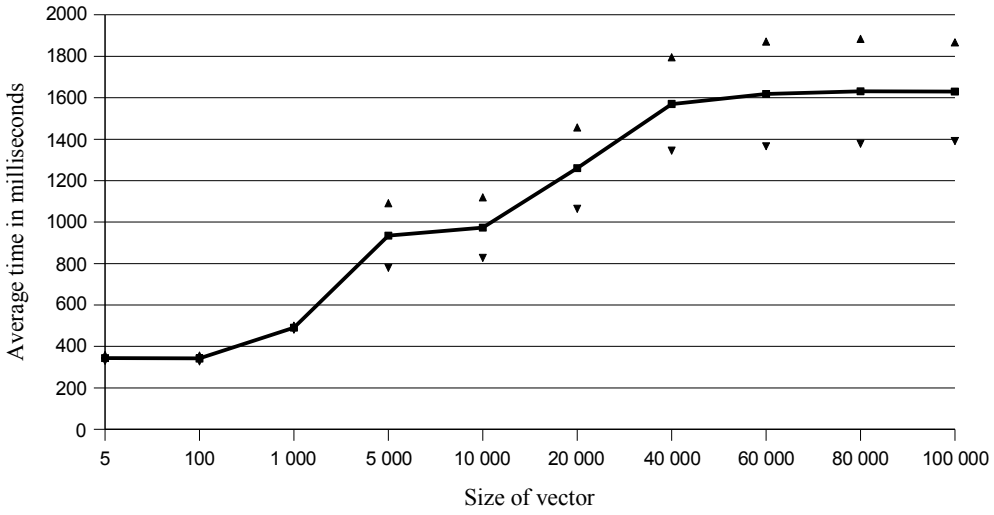

Figure 12: Timings for computing the scalar product

The diagram in Figure 12 is based on the timings in Table 11 and shows how the computation time increases with growth of the vector. We see that our implementation of scalar product computation scales very well. Computing the scalar product of two 100 000-element vectors takes about five times as long as computing the scalar product of two five-element vectors. Figure 12 also shows how the standard deviation affects the computation time for each size of vector. We see that starting from 5 000-element vectors the deviation increases considerably. This effect can be attributed to the large size of the network messages exchanged during the protocol. Processing the vectors in smaller batches could decrease the standard deviation, but the additional overhead of more protocols sent might also provide a negative effect. Further experiments have to be conducted to get conclusive results.

Since the local computations in our system are simple and efficient, most of the time is spent in exchanging network messages. This hypothesis is supported by the data in Tables 1-10 which shows no significant increase in the timings for computing the sum of the products as the size of the vectors increases considerably. This means that the best path for optimising our system is to reduce the number of rounds in the protocols.

In 2006 Yang *et al* implemented secure scalar product computation between two parties based on homomorphic encryption [YWS06]. The running times for their implementation are linear in the size of the input vectors. They provide experiment results for two separate implementations—one, that precomputes the

encryptions for the vector and the other that does not do any precomputation. Without precomputation their implementation computes the scalar product of two 100 000-element vectors in 17 minutes. With precomputation the running time of the actual protocol is reduced to five seconds, but the encryptions have to be computed beforehand and stored securely. The running times for their implementation are linear in the size of the input vectors which means that if the vector is twice as large it takes two times as long to complete the computation.

In conclusion, our implementation of the share multiplication protocol is efficient and suitable for use in real data processing applications. As expected, the implementation is more efficient when used on vectors of values not single values.

6.3 Evaluating the greater-than predicate

To evaluate the greater-than predicate we generate two values and find out, which one was greater. We measure the time taken to load the data and to evaluate the predicate separately. The results of the experiment are given in Table 6.3.

Operation	Minimum	Maximum	Mean	Median	Standard deviation
Load	36	72	48.7	48	2.84
GT	923	2004	999.95	976	78.02
Total	971	2050	1048.65	1031	78.11

Table 12: Timings for evaluating the greater-than predicate in 10 000 iterations

According to our results evaluating greater-than predicate in SHAREMIND takes an average of one second that is computationally quite expensive. The main reason for the cost in time is the complexity of the bit extraction sub-protocol that currently requires 38 rounds. The currently implemented protocol is suboptimal as there exist protocols with less rounds, like the one used in FairPlay [MNPS04].

A two-party version of the greater-than evaluation is presented by Malkhi *et al* in their FairPlay implementation [MNPS04]. Their implementation evaluates the greater-than predicate in 1.25 seconds. Our solution is faster, but only marginally. For developing real-life applications we need a more efficient protocol.

7 Conclusion

In this thesis we present a framework for performing secure multiparty computations. Our main goal was to build a model for privacy-preserving computations that could be easily and efficiently implemented in software. We achieved this goal and in this work we present both the developed theoretical framework and the practical implementation.

Our solution uses three computing parties for efficiency reasons—the known two-party computation protocols are computationally inefficient while adding more parties requires more communication and is economically more expensive. The data values in our framework are elements of $\mathbb{Z}_{2^{32}}$. We use the additive n-out-of-n secret sharing scheme for distributing the data between the parties and provide protocols for performing basic operations on shared data.

We present protocols for addition, multiplication by scalar, multiplication, converting shares from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$, extracting bits of a value and performing the greater-than comparison. We show that these protocols are correct and prove that they are perfectly secure in the honest-but-curious model with up to one corrupt party in the system.

Our main result is the implementation of our framework in a software platform called Sharemind. We built Sharemind to show that our theoretical approach is practical and feasible. The software is written by using the GNU C++ compiler and uses the RakNet networking library for communication. Sharemind is a cross-platform application and has been tested on Linux, Mac OS X and Microsoft Windows. The communication between parties uses the UDP protocol in the Internet protocol suite. The source code of the prototype version of Sharemind is presented together with this thesis.

The implementation consists of the miner software and the controller library. The miner software is run as a server that contains protocol implementations and provides a computational environment for data processing algorithms. The controller library provides a programming interface to use the services provided by the miners. The controller library allows the user to easily implement data processing applications by translating the users' commands to protocol invocations for the miners. The most immediate benefit of this library is the decrease in time needed to develop prototypes for privacy-preserving data processing algorithms

We have developed a number of testing applications to show how the platform operates. This thesis also presents experiment results from timed executions of these testing applications that show the feasibility of our approach. We tested a Sharemind-based implementation of a scalar product computation algorithm and timed the results. The implementation scales very well and provides performance sufficient for real-life computations. The results for greater-than predicate were less impressive and show room for improvement.

8 Kuidas teha turvaliselt arvutusi ühissalastatud andmetega

Dan Bogdanov

Magistritöö (40 AP)

Kokkuvõte

Käesolevas magistritöös kirjeldame turvaliste mitme osapoolega arvutuste läbiviimiseks loodud raamistikku. Meie peamine eesmärk oli luua raamistik, mille abil saab kirjeldada andmete privaatsust säilitavaid arvutusi ning mis oleks kergesti tarkvaras realiseeritav. Privaatsuse säilitamine on isikustatud andmete töötlemisel oluline, kuid üldtunnustatud tehnilist lahendust selle jaoks veel loodud ei ole.

Töös esitame raamistiku, mis kasutab turvalisi ühisarvutusi ja ühissalastust turvatud arvutuskeskkonna loomiseks. Arvutusteks otsustasime kasutada kolme osapoolt, sest teadaolevad kahe osapoolega protokollid on ebaefektiivsed, samas liiga paljude osapoolte rakendamine suurendab protokollide keerukust ning tõstab süsteemi hinda.

Meie raamistik toetab andmeid, mis on esitatavad hulga $\mathbb{Z}_{2^{32}}$ elementidena. Me kasutame andmete osakuteks jaotamisel aditiivset n-seast-n ühissalastusskeemi. Töö sisaldab ühisarvutuse protokolle, mille abil saab ühissalastatud andmetega teha arvutusi ilma algseid väärtusi taastamata. Esitame protokollid andmeosakute liitmiseks, omavaheliseks ja skaalariga korrutamiseks, teisendamiseks hulgast $\mathbb{Z}_2$ hulka $\mathbb{Z}_{2^{32}}$, bittideks jaotamiseks ning suurem-kui võrdlemiseks. Me näitame, et protokollid on korrektsed ja täielikult turvalised ausas-kuid-uudishimulikus turvamudelis, kus on lubatud üks korrumpeerunud osapool.

Raamistiku praktilisuse tõestamiseks oleme loonud tarkvaraplatvormi nimega Sharemind, mis realiseerib alamosa raamistiku võimalustest. Programmikood on kirjutatud GNU C++ kompilaatorit ja võrguliikluse teeki RakNet kasutades. Sharemind töötab erinevate operatsioonisüsteemidega ning me oleme seda katsetanud Linuxi, Mac OS X ja Microsoft Windowsi keskkonnas. Koos magistritööga esitame ka tarkvara lähtekoodi.

Tarkvaralahendus koosneb andmekaevandaja rakendusest ning juhtimisteegist. Andmekaevandaja rakendus, mille sees on realiseeritud raamistiku protokollid, töötab serverina ning pakub andmetöötlusrakendustele vajalikku arvutuskeskkonda. Juhtimisteek annab programmeerijale käsustiku, mida kasutades saab andmekaevandajate teenuseid kergesti kasutada. Teek tõlgib programmeerija juhised andmekaevandajatele arusaadavateks protokolliväljakutseteks.

Töö käigus loodi mitmeid testrakendusi tarkvaraplatvormi võimaluste tutvustamiseks. Magistritöös esitatakse ka eksperimenditulemused, mis näitavad süsteemi

jõudlust. Me testisime SHAREMIND'i abil ehitatud skalaarkorrutise arvutamise algoritmi realisatsiooni ja mõõtsime programmi tööaega. Meie realisatsioon skaleerub väga hästi ja on piisavalt kiire reaalsetes rakendustes kasutamiseks. Suurem-kui võrdluse arvutamine on aeglasem ning vajab efektiivseks rakendamiseks kindlasti täiendamist.

Autor soovib tänada oma juhendajaid, kelle pühendumus ja põhjalikkus olid hindamatuks panuseks käesoleva magistritöö valmimisele.

References

[BDJ+06] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography*, volume 4107, 2006.

[Ben86] Josh Cohen Benaloh. Secret sharing homomorphisms: Keeping shares of a secret sharing. In *Proceedings on Advances in Cryptology—CRYPTO '86*, pages 251–260, 1986.

[Bla79] George R Blakley. Safeguarding cryprographic keys. In *Proceedings of AFIPS 1979 National Computer Conference*, volume 48, pages 313–317, 1979.

[Bog07] Dan Bogdanov. Source code of SHAREMIND. Published online at http://www.sourceforge.net/projects/sharemind/, 2007.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10, 1988.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[CD04] Ronald Cramer and Ivan Damgård. Multiparty computation, an introduction. Course Notes, 2004.

[DA00] Wenliang Du and Mikhail J. Atallah. Secure remote database access with approximate matching. In *First Workshop on Security and Privacy in E-Commerce, Nov. 2000.*, 2000.

[Dam02] Ivan Damgård. Secret sharing. Course notes, 2002.

[Des88] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, 1988.

[DF89] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 307–315, 1989.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Third Theory of Cryptography Conference, TCC 2006*, volume 1880 of *Lecture Notes in Computer Science*, pages 285–304, 2006.

[Gol04] Oded Goldreich. *Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[Kil05] Mati Kilp. *Algebra I*. Estonian Mathematical Society, 2005.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In Proceedings of the 13th USENIX Security Symposium (2004), pp. 287-302., 2004.

[MS81] R. J. McEliece and D. V. Sarwate. On sharing secrets and reed-solomon codes. *Communications of the ACM*, 24(9):583–584, 1981.

[NS07] Janus Dam Nielsen and Michael I. Schwartzbach. A domain-specific programming language for secure multiparty computation. To appear in PLAS 2007 — the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, 2007.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[Sho00] Victor Shoup. Practical threshold signatures. *Lecture Notes in Computer Science*, 1807:207–220, 2000.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science*, pages 160–164, 1982.

[YWS06] Zhiqiang Yang, Rebecca N. Wright, and Hiranmayee Subramaniam. Experimental analysis of a privacy-preserving scalar product protocol. *Comput. Syst. Sci. Eng.*, 21(1), 2006.