DISSERTATIONES
MATHEMATICAE
UNIVERSITATIS
TARTUENSIS
113

JAAK RANDMETS

Programming Languages for Secure Multi-party Computation Application Development

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

113

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
113

JAAK RANDMETS

Programming Languages for Secure Multi-party Computation Application Development

1632 UNIVERSITY OF TARTU
Press

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

Dissertation is accepted for the commencement of the degree of Doctor of Philosophy (PhD) on April 25, 2017 by the Council of the Institute of Computer Science, University of Tartu.

The public defense will take place on June 5, 2017 at 15:15 in J. Liivi 2-405.

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

European Union
European Social Fund

Investing
in your future

ISSN 1024-4212
ISBN 978-9949-77-432-6 (print)
ISBN 978-9949-77-433-3 (PDF)

University of Tartu Press
www.tyk.ee

Contents

ABSTRACT

Secure multi-party computation is a technology that allows computation on private data without revealing any secrets. If the data is processed in an encrypted form, then this means that at no stage of the computation is the data ever decrypted. As a theoretical concept the technology has been around since the 1980s, but the first practical implementations arose a bit more than a decade ago. Since then, secure multi-party computation has been used in practical applications, and has been established as an important method of data protection.

When developing any application, the choice of programming language is dictated by many considerations: personal preference, performance requirements, hardware characteristics, and security constraints. Similar criteria apply when choosing a language for developing an application that makes significant use of secure multi-party computation methods. Unfortunately, regular general-purpose languages are not perfectly suitable. First, there is the need to tightly control what data is kept private and what can be revealed. Second, parallelism needs to be exploited even at the very basic level in order to make performance acceptable. In this thesis we present SecreC 2—a high-level programming language meant for secure multi-party computation algorithm development that satisfies these goals.

High-level secure multi-party computation applications invoke operations that execute network communication protocols. Developing applications that call these protocols is a different task from developing the protocols themselves. Secure multi-party platforms can be viewed as distributed computers offering instructions that can be invoked to perform general computation. When specifying and implementing secure multi-party computation operations, we can take similar approaches as to how computer hardware is designed. For designing low-level electronic and digital logic circuits highly-specialized hardware description languages are used. In this thesis we introduce a new domain-specific programming language for specifying low-level multi-party computation protocols.

We claim that domain-specific programming languages, such as the ones we introduce in this thesis, enable the building of secure multi-party applications and frameworks that are at the same time usable, efficient, maintainable, trustworthy, and practically scalable.

CHAPTER 1

INTRODUCTION

1.1 Secure multi-party computation

Secure multi-party computation (SMC) is a set of cryptographic techniques that allows participants to securely process data. It is a distributed computation model where some of the participants may supply data and some are involved in computation that may involve multiple interactive communication rounds. During the computation, at no stage are secret values revealed. Theoretical protocols for SMC appeared in the 1980s but practical implementations surfaced much later in the early 21st century. The field is still rapidly developing.

SMC has been around for a while but very few practical applications use the technology. Performance results for many prototype applications have been published but none of them process real data for a non-academic purpose. The first real application of SMC was the Danish sugar beet auction in 2008 [24]. The largest scale application has thus far been a social study conducted in 2015 [17] where tax payments from the Estonian Tax and Customs Board were linked with higher education events from the Ministry of Education and Research. The study processed ten million privacy sensitive tax records and half a million education records in a privacy-preserving manner.

For SMC to become more commonplace the technology itself and the frameworks have to improve. Unfortunately, when it comes to usability, performance, and scalability, SMC has a long way to go. Currently only cryptography experts develop SMC applications and these perform too poorly for many practical problems.

1.2 Programming secure multi-party computation

We vision that SMC applications should be as easy to develop for regular programmers as public computation application are. This thesis looks to improve the state of the art in this regard by offering a better high-level application development language for SMC.

SMC applications are usually developed using specialized programming languages [87, 58, 59, 84, 85, 91, 118, 99, 53]. General-purpose languages are not perfectly suitable for SMC. One problem is that in SMC applications we want to control the flow of information very carefully. At no stage should private information be revealed unless it is explicitly intended by the programmer. Information flow control is usually achieved at the level of types: values are classified into *public* and *private*. Public data can be implicitly converted to private data but conversion in the other direction has to be explicit. While type systems of some general-purpose programming languages are powerful enough to restrict information flow, they can be quite difficult to use in this regard.

Another aspect where developing SMC applications differs from regular public applications is performance. Even the best SMC schemes are still many orders of magnitude slower than normal computations. In many schemes individual operations, such as multiplying two secure numbers, take multiple network communication rounds. This means that a single secure integer multiplication can take tens of milliseconds due to network latency whereas a public multiplication can be performed in nanoseconds[1]. The difference in efficiency can be seven or even more orders of magnitude. Fortunately, the situation can be drastically improved by executing many instructions at the same time. Because computer networks are optimized for high-bandwidth throughput, we can perform millions if not tens of millions of multiplications per second, reducing the overhead compared to public computations to only a few orders of magnitude. This means that for SMC applications to be reasonably fast, parallelism has to be exploited at a very basic level and application development languages have to facilitate this kind of coding style.

SMC platforms can be viewed as distributed computers that offer instructions that can be called to perform general computation. It is extremely important that these instructions are efficiently implemented. For SMC platforms that have a wide range of specialized primitive operations, this poses an implementation and maintenance problem. Often the low-level protocols are written in a general-purpose language such as C++ or Java. This kind of highly-optimized low-level networking code is very difficult and error-prone to write. It is hard to trust that these implementations are correct and do not contain security errors. In this thesis we introduce a new *domain specific language* (DSL) for specifying low-level SMC protocols. The language aims to make the development of complicated protocols easier and to increase trust that the operations are correct and secure.

[1]This is ignoring the possibility of multiple cores, instruction-level parallelism and SIMD operations.

1.3 The claims of this thesis

We claim that the use of domain-specific languages enables building secure multi-party applications and frameworks that are at the same time usable, efficient, maintainable, trustworthy, and practically scalable.

We demonstrate *usability* by SecreC 2 [20] applications that have been developed. SecreC 2 is a high-level SMC applications development language designed and implemented by the author of this dissertation. While there are many other languages with similar goals, very few of them provide comparable reusability and none have been tested in large-scale real-life applications. SecreC 2 is by far the most used SMC language with the largest standard library to make application development easier.

We say that SMC framework is sufficiently usable if non-trivial applications can be developed with its help in a reasonable effort by someone who is not knowledgeable in secure computing or cryptography in general. It is not possible to prove so informal claim but we hope that a reader is convinced by the many SecreC 2 code examples. Many applications and prototypes have been developed using Sharemind on top of SecreC 2 language.

We demonstrate that *efficiency* is enabled by both SecreC 2 and the protocol DSL. The DSL facilitates building efficient primitives: Sharemind is one of the most competitive generic platforms when it comes to performance. We also show that SecreC does not restrict the programmer when it comes to performance: the language facilitates SIMD vectorization, enables the programmer to make trade-offs between performance and security, and, if all else fails, allows the programmer to implement specialized protocols in C++ or the DSL and invoke them from SecreC. We also perform extensive benchmarking to demonstrate efficiency. We can show speedups of orders of magnitude over many of our previous protocols.

We achieve *maintainability* via the use of the protocol DSL. Protocols can be implemented in a compositional style where larger protocols call simpler ones. When some of those simpler protocols are modified the larger ones require no changes. Protocol implementers do not have to think as much about manual optimizations. Changes to the underlying platform (such as the networking layer) do not require modifications to protocols.

For SMC framework to be maintainable new functionality has to be easy to add. The system must not be fragile in the sense that modifying existing functionality should not need overarching changes to the entire system. To demonstrate maintainability, we show how we re-implemented the entire protocol stack of Sharemind, and later modified many of the underlying protocols and implemented new optimizations. These modifications were easy to make which instills confidence that we have made correct language design choices.

Language-based tools enable *trustworthiness* in a multitude of ways: the type system of SecreC catches trivial misuses but also enables program analysis for more fine-grained checks. Additionally, SecreC 2 has been designed with security in mind and does not contain loopholes that let high-level application developer execute unsafe code. The protocol DSL guarantees the security of the underlying protocols via the use of a static analysis tool. The majority of Sharemind protocols have now been verified with the tool. The formalization of SecreC instills extra confidence: we know that the program is secure if the underlying protocols are secure and remain secure under composition.

For SMC system to be *scalable* it must be able to process realistic data volumes in the scale of at least 10^7 records. The system must also tolerate long up time to be able to execute non-trivial algorithms on so much data. We postulate that scalability, while not impossible to achieve and surpass without language-based tools, is enabled by the languages we have introduced. We demonstrate both vertical and horizontal scalability by extensively benchmarking individual operations. We see that Sharemind platform is capable executing large applications that process tens of millions data records.

1.4 Outline and the author's contributions

The author has designed and implemented a high-level programming language called SecreC 2 for developing SMC data mining algorithms. The usefulness of the language has been validated through many in-house and third-party applications and prototypes that rely on the language. The author has also been heavily involved in the development of the standard library components of SecreC 2.

The author has designed and implemented a domain-specific language for implementing low-level SMC protocols. The design choices have been validated, firstly, by implementation of protocols, and performance results demonstrating significant speedups over previous implementation, and, secondly, by added value in the form of automatic optimizations and security proofs.

Chapter 2 gives an overview of secure (multi-party) computation and looks at some of the most common ways SMC can be implemented. The overview of secure computation covers Yao's garbled circuits, homomorphic encryption, trusted execution and secret sharing. We also give a short overview of how the security of secure computation is usually defined via the concept of universal composability.

Chapter 3 describes the SecreC 2 language. The overview is very informal and is aimed at developers who want to see the language in action. The tutorial is largely based on code examples and does not assume the reader to be closely familiar with secure computation. We start by giving a gentle introduction to how SecreC 2 is used for secure computation and introduce the concept of protection domains. We follow by describing its more complex features like array processing capabilities and domain-polymorphism.

Because efficiency is very important in SMC, we also give some tips on how to write SecreC code that performs well. This largely focuses on data parallelization, exploiting the properties of the specific schemes, and optimizing algorithms by revealing information when it is deemed acceptable. Most of these techniques apply to any SMC application development framework. However, the examples demonstrate the capabilities of SecreC and are useful for learning purposes.

We also discuss the implementation of the SecreC compiler and its integration with the Sharemind framework. Finally, we discuss some upcoming and possible future improvements of the language.

Chapter 4 gives formal semantics to the core of the SecreC 2 language. We present the syntax of the core language, specify its type system and formalize its dynamic behavior. Using the language semantics we can state the security theorem: if the underlying protocols are universally composable then SecreC programs are secure. Finally, we formalize how to convert domain-polymorphic programs to monomorphic programs and show that the translation preserves semantics and security. This chapter is based on a previous publication of the author [20]:

- Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014. pp. 53–65 (2014).

The author of this work designed SecreC 2 and was the main implementer of its compiler. The author also formalized the core of SecreC 2 in collaboration with Peeter Laud.

Chapter 5 introduces the protocol DSL. We first give a gentle overview of the language, heavily relying on example code. We use the protocol DSL to implement a large set of additive three-party secret sharing protocols. While not a complete reference, the language overview section is intended for the users of the language. The author of this work designed the language, designed its intermediate form, was the principal developer of the DSL compiler and evaluated the performance of all the protocols.

We also discuss the implementation of the language and how it integrates into the Sharemind framework. We describe its intermediate arithmetic-circuit form, and show how we use it for security analysis. We benchmark the protocols written in the DSL very thoroughly and compare their performance against our previous protocol set. Finally, we discuss possible future directions.

This chapter is based on the previous publications of the author [77, 64]:

- Laud, P., Randmets, J.: A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1492–1503 (2015).
- Kerik, L., Laud, P., Randmets, J.: Optimizing MPC for robust and scalable integer and floating-point arithmetic. In: Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. pp. 271–287 (2016).

The author of this work designed the protocol DSL and formalized its data type system and dynamic behavior. The author also designed the intermediate protocol representation. The author was the principal developer of the protocol language compiler and contributed to the development of additive three-party integer protocols. All performance benchmarks have been performed by the author.

Chapter 6 gives an overview of applications and prototypes that have used and benefited from either SecreC 2 or the protocol DSL. For each application we describe to what degree SecreC was used and if the application benefited from the protocol language.

Chapter 7 gives an overview of the most notable programming languages that are used in SMC and are related to our work. We have included both languages for low-level protocol specification and languages for high-level application development. We have categorized them roughly into ones that are targeted for Yao's garbled circuits, ones that are targeted for secret sharing, and ones that allow a mixture of different SMC techniques to be used.

CHAPTER 2

PRELIMINARIES

2.1 Secure computation

The goal of secure multi-party computation (SMC) is for parties to jointly compute a function over their inputs while every party keeps their inputs private. This is formalized as *secure function evaluation* where n parties compute a function

$$f(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n) \ ,$$

such that each party $\mathcal{P}_i$, where $1 \leq i \leq n$, provides its input x_i and learns its output y_i. Usually we want the computation process to be *input private* meaning that a party can only learn about other parties' inputs that which can be derived just from the party's own input and output. It is also desirable for the computation scheme to be *correct*.

There are multiple different ways of implementing secure function evaluation. Each scheme and its variations offer different trade-offs between security, performance and usability. In the following we will give a short introduction to four secure function evaluation schemes: garbled circuits, (fully) homomorphic encryption, secure hardware, and secret sharing.

2.1.1 Roles in secure computation

We classify secure computation participant into three roles to help explain how they manipulate data and what is visible in plain text to which party. Some of the participants act as *input parties*. They secret share or encrypt data and send the hidden values to *computing parties* who perform operations without revealing private inputs. After that secret shared or encrypted results are sent to *result parties* who can declassify or decrypt the resulting values.

A participant can have multiple roles. A computing party can act as an input party if it has data to share data and also result party if it needs to see the results.

One such example is the classic millionaires' problem where two participants want to learn which of them is richer. Both participants supply data, both are involved in computing and both learn the public result. In some SMC deployments roles might not overlap at all. For example in survey systems anonymous participants can supply data through web interface, computing parties perform statistical analysis on their hidden inputs and only statistician sees the final result. In her dissertation [60] Liina Kamm discusses participant roles in more detail and gives examples of various different deployment models.

2.1.2 Garbled circuits

The garbled circuits (GC) approach was introduced by Yao [113, 114]. In a GC scheme two parties securely compute $f(x, y)$ where f is a known functionality represented as a Boolean circuit. The protocol operates asymmetrically. Party $\mathcal{P}_1$ is known as the *garbler* and $\mathcal{P}_2$ is known as the *evaluator*. They respectively provide inputs x and y as bit strings and $\mathcal{P}_1$ learns the output. Intuitively, the garbler encrypts the Boolean circuit $f(\cdot, \cdot)$, and sends the garbled truth tables, and keys corresponding to x to the evaluator. The evaluator then uses an *oblivious transfer* (OT) protocol to obtain the keys corresponding to its input to decrypt the garbled circuit and evaluate the encrypted $f(x, \cdot)$ on y. During this process the evaluator remains completely oblivious and only learns the intermediate random keys. Finally, the evaluator sends the keys corresponding to the output wires to the garbler who knows which key corresponds to which value. In this setting both parties supply input, both are involved in computation and $\mathcal{P}_1$ acts as the only result party.

The gates are garbled by looking at all the possible inputs and encrypting the output key with the keys that correspond to the input values. More formally, for each wire in the circuit, denoted with label i, the garbler encodes inputs 0 and 1 as randomly generated keys k_i^0 and k_i^1. Let g be a gate with input wires i and j, and output wire w. The gate g is garbled as follows: for every combination of input bits b_1 and b_2 and the corresponding output bit b we encrypt the output wire key k_w^b using the combined key $(k_i^{b_1}, k_j^{b_2})$. This way, only by knowing the appropriate input keys is it possible to learn the output key. In the case of binary gates a total of four ciphertexts are stored, each corresponding to one combination of input values.

The garbled circuits approach has some very attractive properties. For one, it is a two-party protocol which simplifies deployment compared to schemes with more parties. In addition to that, evaluating arbitrarily large circuits requires only a few communication rounds. Third, the scheme is generic and can evaluate any Boolean circuit. Boolean circuits are a very convenient format. Much research has been done on how to efficiently represent functions with Boolean circuits.

However, there are also a few challenges to overcome. Boolean circuit representations of functions are often very large, and garbling and evaluation are computation-heavy operations. If the whole circuit is constructed and stored in the memory then evaluating it can require excessive amounts of memory. GC schemes are also not as communication-efficient as some secret sharing schemes are. For example, addition does not require any network communication in additive secret sharing schemes but requires significant communication with the (Boolean) GC approach.

Scalability challenges have been tackled by VMCrypt software architecture [86] and by Huang et al. [57] with the so-called pipelining approach. In this approach, instead of garbling the entire circuit, parts of circuits are sent to the evaluator immediately when they are ready. This way, the evaluator does not have to wait for the whole circuit to be garbled and can start with evaluation and oblivious transfers as soon as possible. Parts of the circuit that have already been evaluated can be discarded to reduce the memory footprint.

Much research has been done on GC protocols since their invention. Communication and computation efficiency has since been greatly improved by many optimizations. Some of these are: the free-XOR technique [68] for allowing XOR gates to be evaluated without networking overhead or much computation overhead, the point and permute technique [90] for reducing the number of decryptions the evaluator has to perform, and garbled row reduction [90] and the half-gate approach [115] that both reduce the size of garbled tables. Many pratical garbled circuit implementations exist now, for example, Fairplay [87], FairplayMP [7], TASTY [53], FastGC [57], ABY [40] and ObliVM [85].

2.1.3 Homomorphic encryption

Homomorphic encryption (HE) allows computations to be carried out on encrypted ciphertexts without revealing the underlying secrets. HE is usually set up in the two-party model where the client holds the secret key and the server is able to perform computations on data encrypted by the client. For example, Paillier [94], lifted ElGamal [45] and Damgård–Jurik [36] cryptosystems are additively homomorphic, allowing to compute arbitrary linear functions on ciphertexts. The method for evaluating quadratic multivariate polynomials (with a limited range) was introduced by Boneh, Goh and Nissim [25] (BGN). All these systems are for evaluating only small classes of functions. Despite this, additively, multiplicatively and *somewhat homomorphic* systems are useful for many specialized applications, such as schemes for tallying encrypted votes [33]. Mostly they are used as a part of other cryptographic primitives.

The first *fully homormorphic encryption* (FHE) scheme was proposed by Gentry [46, 47]. Like BGN, the scheme alone was initially only able to evaluate low-degree multivariate polynomials. This was due to noise added by operations on encrypted texts. When sufficiently many operations are performed, the level of noise becomes so great that decryption is no longer possible. To lower the noise Gentry proposed *bootstrapping* where ciphertexts are encrypted for the second time, and the underlying initial cryptotext is decrypted homomorphically. The inner decryption requires that the scheme can evaluate its own decryption function. Gentry showed that after bootstrapping, the scheme could safely apply one extra operation. The resulting scheme is truly fully homomorphic, being able to evaluate arbitrarily complex functions on ciphertexts.

Initial FHE implementations were impractically slow for even basic bit operations. In [48] it is reported that, the re-encryption (bootstrapping) step takes up to 30 minutes for large security parameters. Modern schemes can evaluate large circuits in reasonable time. For example, later works, such as [42], report less than a second for every bootstrapping step on a personal computer. Progress has been fast but not all challenges have been resolved. Efficiency is not yet competitive with other SMC schemes. Secret keys are still hundreds of megabytes large, and security parameters are numerous and difficult to understand.

2.1.4 Trusted execution environment

The *trusted execution environment* (TEE) is a secure area of a computer, or another electronic device (smart phone, tablet, television), that guarantees that the code and data inside are protected with respect to confidentiality and integrity. Modern incarnations of secure hardware include Intel SGX, ARM TrustZone and AMD Secure Processor. An overview of the research on secure hardware applications and security guarantees can be found in [96]. Applications of TEEs and secure hardware include secure authentication to protect the user of the device against malicious apps, secure key storage, digital rights management, secure boot, securing financial transactions, and even preventing cheating in online games.

Trusted computing in general has been subject to controversy by digital rights groups such as the Electronic Frontier Foundation[1] and the Free Software Foundation[2], but also by security experts[3]. One of the criticisms is that trusted computing can be used to heavily restrict users' freedoms by securing devices against their owners. For example, a device can strictly limit which documents it can access and what applications can be used.

Remote attestation is a process that allows verification of a running software configuration on a device. *Provisioning* is a process of securely sending secrets and code to the TEE of the target device.

In SMC, secure hardware can be used to improve the security of existing schemes by executing them in isolation. This provides protection against a compromised

[1]`https://www.eff.org/issues/trusted-computing` (April 2017)

[2]`https://www.gnu.org/philosophy/can-you-trust.en.html` (April 2017)

[3]`https://www.schneier.com/crypto-gram/archives/2002/0815.html#1` (April 2017)

operating system and allows parties to validate each others' software integrity via remote attestation. This can improve security guarantees of SMC schemes that are otherwise secure only against honest-but-curious adversaries. In [109] authors envision a cloud-assisted malware checking service. To protect users' privacy, they use a private membership test where users communicate with trusted hardware via secure channels.

2.1.5 Secret sharing

Secret sharing is a set of methods that allows for a secret value x to be split between n participants as $x_1, \ldots, x_n$ so that each party $\mathcal{P}_i$ gets a random-looking *share* x_i. The secret value x can only be reconstructed when at least t ($t \leq n$) of the shares are combined. Schemes with n participants, of which a threshold of t are required to reconstruct the original value, are called (t, n)*-threshold schemes*. The concept was independently discovered by Shamir [104] and Blakley [8].

Secret sharing based secure computation schemes are flexible in the sense that they easily scale to arbitrary number of input and output parties. Anyone can easily secret share their data and send it securely to computing parties. Similarly, results are easy to share with output parties through secure communication channels. Usually both secret sharing and reconstruction are inexpensive operations requiring only random number generation and small amount of network communication.

In [104] Shamir proposed a t-out-of-n secret sharing scheme based on the fact that t points are required to uniquely determine a $(t-1)$-degree polynomial. To share a secret value x, a random $(t-1)$-degree polynomial f over a finite field $\mathbb{Z}_p$ such that satisfying $f(0) = x$ is chosen. Shares are computed as $x_1 = f(1), x_2 = f(2), \ldots, x_n = f(n)$. Given t of those points, the polynomial f can be reconstructed via Lagrange interpolation and the secret is reconstructed as $f(0)$. For general secure computing share construction and reconstruction protocols are not enough. Chaum, Crépeau and Damgård [31] showed that general computation that is information theoretically secure can be done on values shared with Shamir's scheme. Shared values are straightforward to add by summing the respective shares locally. Multiplication is more involved and requires parties to communicate with each other.

Additive secret sharing

In this section we give a brief overview of the n-out-of-n *additive secret sharing* scheme. We present the scheme in more detail because much of this thesis is based on SMC built on 3-out-of-3 additive and bitwise additive schemes. To share an integer $x \in \mathbb{Z}_N$ additively between n parties we pick $n-1$ random integers from $\mathbb{Z}_N$. All but one of the participants are given a randomly generated value and one of the parties is given the secret x minus the sum of the random values

$$\begin{array}{rcl}(x_1, \ldots, x_{n-1}) & \overset{\$}{\leftarrow} & \mathbb{Z}_N \times \ldots \times \mathbb{Z}_N \\ x_n & \leftarrow & x - \sum_{i=1}^{n-1} x_i \mod N \ .\end{array}$$

The original value can be reconstructed by summing all of the shares. Note that the additive scheme can also be instantiated when N is 2. This is a useful case as it allows for secret sharing of single-bit integers. We can lift this scheme to operate on bit vectors of length k, yielding bitwise or *XOR sharing* over $\mathbb{Z}_2^k$.

In the following we will use $[\![x]\!]$ to denote a secret shared value that reconstructs to x. A shared value $[\![x]\!]$ is an n-tuple with each computing party holding exactly one component of the tuple. The individual shares of $[\![x]\!]$ are denoted with $[\![x]\!]_1, \ldots, [\![x]\!]_n$ and individual values $v_i \in \mathbb{Z}_N$ can be combined to form a shared value $(v_1, \ldots, v_n) = [\![\sum_{i=1}^{n} v_i]\!]$. Reconstruction of x is as easy as computing $\sum_{i=1}^{n} [\![x]\!]_i$. Binary operations on shared data are denoted as $[\![x]\!] \star [\![y]\!]$ and may use communication protocol between parties. For example, addition is denoted with $[\![x]\!] + [\![y]\!]$ and multiplication with $[\![x]\!] \cdot [\![y]\!]$. Addition can be performed locally in the additive scheme:

$$[\![x]\!] + [\![y]\!] = ([\![x]\!]_1 + [\![y]\!]_1, [\![x]\!]_2 + [\![y]\!]_2, \ldots, [\![x]\!]_n + [\![y]\!]_n) \ .$$

One must be careful with the notation. We write $[\![x + y]\!]$ to denote a secret shared value that reconstructs to $x + y$ and *not* the addition operation between shared values. It is correct to say that the aforementioned scheme for addition implements $[\![x + y]\!]$. Also, note the placement of indices: by $[\![\vec{x}_i]\!]$ we do not mean i-th share of the vector $\vec{x}$ but instead secret shared i-th component of the vector $\vec{x}$.

Multiplication of additively secret shared values requires network communication. Let us consider only the three-party case. One possible implementation is based on the idea that

$$\begin{array}{rl} & (x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\ = & (x_1 y_1 + x_1 y_3 + x_3 y_1) \\ + & (x_2 y_2 + x_2 y_1 + x_1 y_2) \\ + & (x_3 y_3 + x_3 y_2 + x_2 y_3) \\ = & \sum_{i=1}^{3} x_i y_i + x_i y_{\mathsf{p}(i)} + x_{\mathsf{p}(i)} y_i \ , \end{array} \tag{2.1}$$

where $\mathsf{p}(i)$ denotes the index previous to i, wrapping back to 3 if $i = 1$. The index of the next party $\mathsf{n}(i)$ is defined in a similar way. Equation (2.1) provides the scheme for the multiplication protocol $[\![x]\!] \cdot [\![y]\!]$ as proposed by Bogdanov et al. [23]. Every party $\mathcal{P}_i$ sends their shares of x and y to the next party $\mathcal{P}_{\mathsf{n}(i)}$ and computes $[\![w]\!]_i = [\![x]\!]_i [\![y]\!]_i + [\![x]\!]_i [\![y]\!]_{\mathsf{p}(i)} + [\![x]\!]_{\mathsf{p}(i)} [\![y]\!]_i$. From Equation (2.1) it is clear that $w = x \cdot y$. This multiplication scheme generalizes to a larger number of parties but does not scale well with respect to communication when $n > 3$.

One alternative implementation of multiplication uses Beaver triples [6]. We say that the triple of shares $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ is a *Beaver triple* when a and b are uniformly randomly generated and $a \cdot b = c$. Assume that parties have such a triple. First, compute $\llbracket d \rrbracket = \llbracket x \rrbracket - \llbracket a \rrbracket$ and $\llbracket e \rrbracket = \llbracket y \rrbracket - \llbracket b \rrbracket$ locally and then reveal both d and e. Revealing these shares does not compromise security as a and b have been randomly generated and thus mask the secret values. Next, each party locally computes

$$\llbracket w \rrbracket_i = \llbracket c \rrbracket_i + e \cdot \llbracket b \rrbracket_i + d \cdot \llbracket a \rrbracket_i + e \cdot d \enspace .$$

It is straightforward to verify that $w = x \cdot y$ for any choice of a and b.

This scheme has various strengths. First, Beaver triples are independent of x and y and can be pre-computed, and stored for future use. The online part of the multiplication protocol is very simple and communication-efficient. Second, the scheme generalizes to an arbitrary number of parties $n > 1$, including the important case when $n = 2$. The Beaver triple scheme, like the additive scheme, also uses a single round of interaction, but when also considering how to generate Beaver triples (such as [40]) it is significantly more communication- and computation-heavy.

There are several practical applications built upon secret sharing based multi-party computation [24, 107, 17]. Many maturing SMC frameworks that use secret sharing exist. Examples of such frameworks include VIFF [35], SEPIA [27], SHAREMIND [22, 11], FRESCO [105], SPDZ [38], ABY [40] and WYSTERIA [100].

The GMW protocol

Goldreich, Micali and Wigderson [50] (GMW) proposed a protocol for evaluating arbitrary Boolean functions securely. Whereas Yao's protocol requires a constant number of rounds and uses oblivious transfers only for the inputs of the garbler, the GMW protocol uses OT for each AND gate and the number of communication rounds depends on the depth of the circuit. For scalar operations, Yao's protocol is more efficient because of the constant round cost, but for large inputs GMW protocol offers better performance due to small communication overhead (see Demmler et al.[39] for performance evaluation of both GMW and Yao on various input sizes).

In GMW protocol inputs and intermediate values are secret shared with each party $\mathcal{P}_i$ holding a share v_i such that value $v = v_1 \oplus v_2$. XOR gates of the circuit can be evaluated locally by XOR-ing the respective shares. AND gates can be evaluated using either 1-out-of-4 OT or precomputed multiplication triples. Note that every layer of the circuit can be evaluated in parallel. For performance considerations it is important to both size- and depth-optimize the circuits, whereas for Yao's protocol only size optimization is relevant.

The GMW protocol has not seen such widespread use as Yao's protocol. However, ABY [40] uses the GMW protocol for Boolean sharing and WYSTERIA [100] uses an n-party generalization of GMW.

2.2 SHAREMIND framework

While the ideas presented in this work are applicable to many different secure computation settings, this work focuses on the SHAREMIND framework. Both of the programming languages created by the author integrate with SHAREMIND framework. In this section we give an overview of SHAREMIND, its history, and its applications.

SHAREMIND [10, 11] is a framework for analyzing and storing data in a privacy-preserving manner. Initially, SHAREMIND was envisioned as a library and privacy-preserving applications had to be written in C++. SHAREMIND offered the interface necessary to invoke protocols on secret shared data. Security was achieved via SMC based on three-party additive secret sharing. The protocol suite and its UC security properties were formally described in [22].

The next major revision of SHAREMIND introduced the first version of the SECREC 1 language by Jagomägis [59] and improved the three-party protocol suite [23]. SECREC 1 simplified the development of privacy-preserving algorithms and made secure computing more approachable to developers with otherwise little knowledge of cryptography. The language also enabled more rapid application development by transitioning away from a low-level assembly-like programming language to one tailored for the specific domain. In SECREC 1 types are classified into public and private. Operations on private data, such as multiplication and divisions, invoke additive three-party protocols. Implicit information flow from private to public is not allowed but can be done explicitly via *declassification*.

The first version of SECREC is limited in its features. It only supports a single data type for 32-bit integers, and private computations are strictly limited to the additive three-party scheme. The language lacks polymorphic procedures. For example, to compute Hamming distance of both private and public data either two versions of the algorithm need to be implemented or, alternatively, the private procedure can be used after public data is converted to private. Neither solution is ideal. The first solution means that the programmer has to duplicate code and the second solution is significantly slower.

Older SHAREMIND applications used SECREC 1, or a low-level assembly language, or even implemented the applications directly in C++.

- Bogdanov, Jagomägis and Laur [13, 14] developed privacy-preserving frequent itemset mining algorithms in SECREC 1.
- Bogdanov et al. [23] implemented k-means clustering in C++ using SHAREMIND as a library.
- Talviste [107] developed a web-based questionnaire application for the Estonian Association of Information Technology and Telecommunications (ITL)

where the data analysis algorithms and various scripts were implemented in SecreC 1.

- Kamm et al. [61] demonstrated that large-scale genome-wide association studies are possible without violating the privacy of individuals. The core algorithms were developed in SecreC 1.
- Bogdanov et al. [12] developed a secure genetic algorithm for the subset cover problem. The algorithm was implemented in SecreC 1.

The third version of Sharemind was a complete rewrite of the platform. The new version allows the use of multiple security schemes and the platform is no longer strictly tied to three computing parties. For example, in [110] a prototype security domain based on Shamir's scheme [104] was implemented, and in [98] a security domain that combined secret sharing and garbled circuits was developed. To further simplify application development and enable the use of the platform's new features the SecreC language has been revised. The second version of SecreC is one of the contributions of this thesis and an overview of it is given in Chapter 3. Performance of primitive integer and floating-point operations has also been greatly improved in the third version of Sharemind. Much of the improvement has resulted from the new protocol implementation language created by the author of this thesis. The protocol language is the second major contribution of this dissertation and a thorough overview of it can be found in Chapter 5. We give overview of applications and prototypes that use SecreC 2 and the protocol language in Chapter 6.

Later, Bogdanov et al. [9] proposed Rmind as an user-friendly tool for secure data analysis. Rmind is syntactically similar to the R programming environment for statistical computing. A major goal of the language is to offer a familiar interface to statisticians who are used to working with tools such as R, SAS or SPSS. Rmind has been used to conduct a real-life secure statistical study [17] for which individual tax records were securely linked with education records.

2.3 Security of secure computation

The additive secret sharing protocols that this work is focused on are secure in the universal composability security framework by Canetti [28]. A protocol is said to be *universally composable* (UC) if it maintains its security properties when run together with any other secure or insecure protocols. Notably, UC protocols can be composed either sequentially or in parallel with other universally composable protocols and the result will also be a UC protocol. This property allows us to take individual UC protocols and build larger protocols or even applications out of them. We have to emphasize that this is a very strong guarantee that allows even people not familiar with cryptography to build secure applications from secure building blocks.

In the following section we will present an informal overview of the universal composability framework. This work does not present a comprehensive overview. For a more formal and detailed presentation please refer to either the original framework [28] or a version simplified for SMC [29]. A slightly gentler introduction to universal composability framework and SMC can also be found in [32].

2.3.1 The universal composability framework

The security of a protocol π is given with respect to some *ideal functionality* $\mathcal{F}$ that describes what the protocol π computes and how it interacts with adversary. In order for π to UC securely implement $\mathcal{F}$ it has to behave like $\mathcal{F}$ in any environment. The real functionality π can have a complicated internal structure and is often composed of simpler protocols that communicate with each other. The ideal functionality $\mathcal{F}$ usually has no internal structure. It only models the correct input-output behavior such that $\mathcal{F}$ and π would have compatible interfaces for providing input and requesting output.

In this framework, a protocol is modeled as a collection of *interactive agents*. An agent is a computational device that sends and receives messages on named ports, and holds an internal state. In a *closed collection*, each port has a single agent writing it, and a single agent reading it. Only closed collections of agents are executable. The *interface* of a collection of agents is the set of named ports occurring in it that lack a reader or writer.

Security is specified against classes of adversaries. Some restriction is necessary because an adversary with unlimited power can break nearly every cryptosystem. In SMC setting an adversary is allowed to corrupt only a limited number of parties. For example, in three-party additive setting, an adversary is allowed to corrupt a single party. There are different kinds of corruption modes. If a protocol is secure against an adversary that is only allowed to eavesdrop on communication (honest-but-curious), we call this protocol *passively secure*. If a protocol is secure against adversaries that may have complete control over the corrupted parties we say that this protocol is *actively secure*. In most cases adversaries are required to be *efficient*, limiting their computational power to that of nonuniform probabilistic polynomial-time algorithms. We denote adversaries with $\mathcal{A}$.

Notice that while the input-output interface of the real-functionality π is compatible with its corresponding ideal functionality $\mathcal{F}$, it is not possible to just replace π with $\mathcal{F}$ in an arbitrary context. The protocol π is generally executed by some number of parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and evaluation can take multiple communication rounds. To make ideal functionality behave in a way that looks similar to the real protocol we need a *simulator* Sim to emulate the internal structure of π without knowing the inputs of $\mathcal{F}$. In other words, the task of the simulator is to make the ideal functionality $\mathcal{F}$ look like protocol π to adversary $\mathcal{A}$. We write $\mathsf{Sim}^{\mathcal{A}}$ to denote that the simulator uses the adversary in a black-box manner. In practice Sim and $\mathcal{A}$ are actors that communicate with each other via ports. We denote the composition $\mathsf{Sim}^{\mathcal{A}}$ (alternatively $\mathsf{Sim}\|\mathcal{A}$) as $\mathcal{A}^{\circ}$ for the adversary in the ideal world.

Protocol inputs are given by an *environment* $\mathcal{Z}$. In fact, the environment supplies inputs to parties, reads all outputs and interacts with the adversary in arbitrary ways. If a party has been corrupted, its inputs are also given to the simulator (otherwise the simulator is not given inputs).

Definition 1 (Universal Composability)**.** We say that a protocol π UC securely implements ideal functionality $\mathcal{F}$ if there exists a simulator Sim such that for any adversary $\mathcal{A}$ the view of $\mathcal{Z}$ in the real world $\mathcal{Z}\|\pi\|\mathcal{A}$ and in the ideal world $\mathcal{Z}\|\mathcal{F}\|\mathcal{A}^{\circ}$, where the ideal world adversary $\mathcal{A}^{\circ}$ is defined as $\mathsf{Sim}^{\mathcal{A}}$, are indistinguishable. The views can be either statistically or computationally indistinguishable.

2.3.2 Security and privacy for SMC protocols

Given the definition of universal composability we can more formally define security, privacy and the class of passive adversaries. Ideal functionality for a (probabilistic) function f is an agent $\mathcal{F}^{f}_{\mathtt{sec}}$ that communicates with participants and the adversary $\mathcal{A}$. We start by describing this interaction.

The agent $\mathcal{F}^{f}_{\mathtt{sec}}$ first receives the set of corrupted parties from the adversary and then sends corruption message to each of the corrupt parties. Next, $\mathcal{F}^{f}_{\mathtt{sec}}$ receives input x_i from party $\mathcal{P}_i$ and if the party is corrupted then x_i is also forwarded to the adversary. For corrupt parties $\mathcal{P}_i$ adversary sends x'_i to $\mathcal{F}^{f}_{\mathtt{sec}}$ and for non-corrupt parties $\mathcal{F}^{f}_{\mathtt{sec}}$ sets $x'_i = x_i$. The agent $\mathcal{F}^{f}_{\mathtt{sec}}$ computes $(y'_1, \ldots, y'_n) = f(x'_1, \ldots, x'_n)$ and sends y'_i to the adversary if $\mathcal{P}_i$ is corrupt. The adversary can either reply with a value y_i or respond with a message $(\mathtt{stop}, j)$ indicating that a corrupt party $\mathcal{P}_j$ stopped execution. In the latter case $\mathcal{F}^{f}_{\mathtt{sec}}$ forwards the stop-message $(\mathtt{stop}, j)$ to all participants and halts. For non-corrupt parties $\mathcal{P}_i$ the agent $\mathcal{F}^{f}_{\mathtt{sec}}$ sets $y_i = y'_i$. Finally, $\mathcal{F}^{f}_{\mathtt{sec}}$ sends y_i to every party $\mathcal{P}_i$ and stops.

Definition 2 (Security)**.** We say that a protocol π *securely implements function* f if π is a UC secure implementation of the ideal functionality $\mathcal{F}^{f}_{\mathtt{sec}}$.

Definition 3 (Passive adversary)**.** We say that $\mathcal{A}$ is a *passive adversary* for the ideal functionality $\mathcal{F}^{f}_{\mathtt{sec}}$ if it defines $x_i = x'_i$ and $y'_i = y_i$ for each party $\mathcal{P}_i$.

Sometimes it is difficult to achieve security against stronger than passive adversaries. Privacy is a weaker property that is easier to achieve but still guarantees that private inputs are not revealed. For a function f let agent $\mathcal{F}^{f}_{\mathtt{priv}}$ work exactly like $\mathcal{F}^{f}_{\mathtt{sec}}$ until computing $(y_1, \ldots, y_n) = f(x'_1, \ldots, x'_n)$. After that the agent stops and does not send the resulting y_i to parties $\mathcal{P}_i$.

Definition 4 (Privacy)**.** We say that a protocol π *privately implements function f* if π UC secure implementation of the ideal functionality $\mathcal{F}^{f}_{\mathtt{priv}}$.

The three-party additive secret sharing based protocol set (Section 2.1.5) provides security against one passive (honest-but-curious) party. Security is in the sense of statistical indistinguishability of the actual execution of the protocol from a simulated one as defined previously. Security implies that the protocol preserves the privacy of honest parties' inputs, and the protocol delivers correct outputs to all honest parties. For passive adversaries, the correctness property trivially holds. We have constructed most of the protocols of Sharemind in such a way that they provide privacy [18, 97], but not correctness, against one maliciously corrupt party.

2.3.3 Hybrid model

Consider an SMC protocol π that uses some other simpler universally composable protocols. We might have a difficult time directly showing that π is also universally composable. To simplify security proofs we often use the *hybrid world* approach. In this model the UC sub-protocols of π are replaced with calls to their ideal functionalities. If we can show that the hybrid world and ideal world are indistinguishable then from the UC property of sub-protocols we also know that the ideal and real world are indistinguishable too.

This is a particularly useful technique when showing UC security of protocols that are purely composed of UC secure sub-protocols and do not perform any communication apart from what the sub-protocols perform. In the hybrid world such protocols collapse down to only local computation, giving us security for free, and we only need to show correctness.

2.3.4 Arithmetic black box

To talk about the security of entire programs we use the concept of an *arithmetic black box* (ABB) first proposed by Damgård and Nielsen [37]. An ABB is an ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ that contains a mapping from public *handles* to private values and allows the parties to perform secure arithmetic with the stored values. Values can be provided to the ABB by the parties themselves or the ABB can be instructed to obtain a value from the outside environment. To perform an operation, the ABB is supplied with the name of the operation, and handles of the arguments and results. Computing parties can also instruct the ABB to publish values stored under the handles. An operation is only performed if a certain number of parties instruct the

ABB to do so. The number of necessary parties depends on the specific security scheme. We denote a concrete implementation of an ABB with π_{ABB}.

The Sharemind framework allows for programs written in the high-level imperative language SecreC to be executed on top of different arithmetic black boxes. Code can be either specific to an $\mathcal{F}_{\mathsf{ABB}}$ or can be more generic. SecreC takes the so-called duck typing approach where generic procedures can be executed on data in any $\mathcal{F}_{\mathsf{ABB}}$ that offers a suitable interface. This facilitates code re-use even across arithmetic black boxes that offer very different sets of operations. If the protocols in the underlying ABB are universally composable, the programs that can be written on top of them also admit strong security guarantees. We will elaborate on the security guarantees of high-level programs in Section 4.6.

CHAPTER 3

THE SECREC PROGRAMMING LANGUAGE

SecreC 2 is a programming language designed for implementing data analysis algorithms on the Sharemind platform. The main goal of the language is to make the process of implementing secure computation algorithms easy for programmers who are not familiar with secure computation or cryptography in general. We achieve this by

- providing an imperative C-like syntax which is ubiquitous in data analysis literature and is generally far more familiar to programmers than other computing paradigms;
- hiding that the program will be possibly run on multiple computers;
- syntactically not differentiating secure computations from regular public computations, when it is reasonable to do so;
- offering generic programming facilities that enable code reuse across public computations and different security schemes; and
- providing a standard library that builds complex functionality on top of simple primitives, thereby allowing security schemes to only implement a small number of basic operations.

However, the language remains domain-specific. We found that general-purpose languages are not a good fit for secure computation. Firstly, in SMC the programmer has to indicate if the data is public or private and can implicitly convert data only from public to private domain but not the other way around. Having only data that is private can incur too great of a performance cost. Secondly, we facilitate intricate array processing in order to make code vectorization easier, which, in turn, makes common performance optimizations easier. Performance has dictated the language design in more than one way: we also do not support any implicit type conversions because it can be extremely costly in non-public domains. There are many more subtle differences relating to general security in programming: array accesses are always dynamically checked, global state is shunned (but still supported), pointers and references are not supported, and so on.

The SecreC 2 language evolved from the experience with the first version of SecreC. The second version is a more mature language that is richer in features. It has more than one integer type, it can facilitate other security schemes in addition to the additive three-party scheme, it has floating-point numbers, it supports (domain) polymorphic functions (whereas the first version of SecreC has strictly monomorphic functions), and array manipulation is more advanced, supporting arbitrary-dimensional arrays. The author of this work designed and implemented all of these features.

In Section 3.1 we describe SecreC to readers who are interested in learning the language or only want a high-level overview. In Section 3.2 we talk how to write efficient SecreC code. Due to very large constant factors and latencies, performance is an extremely important consideration in secure computing. While we talk about the SecreC language, the techniques are generally applicable to the secure computation setting. In Section 3.3 we give an overview of the implementation of SecreC and talk about how the language fits into the Sharemind framework. Finally, in Section 3.4 we talk about ongoing work on the language and possible future directions.

3.1 Language overview

3.1.1 Familiar syntax

A major goal of SecreC is to offer a familiar syntax. It is infeasible to expect the users to be fluent in functional languages or complex type systems that provide strong security guarantees. For this reason SecreC has a C-like syntax which is ubiquitous amongst popular programming languages. In fact, just by looking at a complete “Hello, World!” program (Listing 3.1) written in SecreC it is impossible to say that the language is somehow unique or related to SMC.

SecreC supports regular public computations the syntax of which is very close to other C-like languages. Arithmetic, bitwise, logical and relational operations are all supported. All integer data types are strictly sized from 8 to 64 bits and include signed and unsigned variants. We also support Booleans, strings, floating-point values and arrays of primitive data types. Basic support for user-defined structures is also available. Listing 3.2 demonstrates simple arithmetics on signed 64-bit

Listing 3.1: “Hello, World!” in SecreC

```
// Simple "Hello, World!" program.
void main() {
    print ("Hello,␣World!");
}
```

Listing 3.2: Integer arithmetic in SecreC

```
void main() {
    int64 x = 25;
    int64 y = - 10;
    print ("x␣+␣y␣=␣", x + y);
    print ("x␣*␣y␣=␣", x * y);
    uint32 z = (uint32) (x - (x / y));
    print ("z␣=␣", 1 + 8 * z);
}
```

and unsigned 32-bit integers. Note that without explicit type conversion when initializing variable `z` the code will not compile and will raise a type error.

A familiar C-like syntax for branching, looping and user-defined functions is available. Function parameters are always passed by value. This includes structures, arrays and non-public values. See Listing 3.3 for an example on defining and calling functions. Note that the `27 :: uint32` syntax is used to specify that the constant 27 is of 32-bit unsigned integer type. We could have also used C-like `(uint)27` but the former guarantees that there is no type conversion happening.

3.1.2 Performing secure computations

SecreC distinguishes between public and private data on type system level. Every type in SecreC has two components: the data type and the security type. When not specified, the security type is implicitly assumed to be public. To declare private data it is not sufficient to state that the data is simply “private”. The programmer has to be explicit what kind of security scheme is used and has to specify the name of the concrete deployment of this scheme. This is because Sharemind facilitates the use of many different security schemes and even the concurrent use of multiple instances of a given scheme.

Having an omitted security domain default to public is not a security issue. The type system guarantees that private data is never implicitly converted to public data. Even if programmer accidentally leaves out a non-public domain a type error will be raised. It is possible to construct an artificial example where this is not true but it would have to involve (indirect) declassifications. Another reason for having the

Listing 3.3: Functions, loops and conditionals in SecreC

```
uint32 nextCollatz (uint32 n) {
    if (n % 2 == 0)
        return n / 2;
    else
        return 3*n + 1;
}
void printCollatz (uint32 n) {
    while (n != 1) {
        print(n);
        n = nextCollatz(n);
    }
}
void main() {
    printCollatz(27 :: uint32);
}
```

Listing 3.4: Secure comparison in SecreC

```
kind shared3p;
domain pd_shared3p shared3p;
void main () {
    pd_shared3p uint x = 3;
    pd_shared3p uint y = 5;
    public bool z = declassify(x < y);
    print ("x␣<␣y␣=␣", z);
}
```

default domain be public is that SecreC programs contain fair share of public code. For example loop counter and array indices are usually public.

To see this in action, consider the code in Listing 3.4 where we first declare a security scheme (or kind) called `shared3p` and then declare a concrete security domain of that kind. In the main function we declare two variables that are in the `shared3p` domain, compare them, reveal the result, and print it. This example only involves computation parties (see Section 2.1.1). The program does not receive input from input parties or produce results to output parties. The computing parties only log the result of declassification.

Private variables are defined similarly to public ones, except the security type is written before the data type. Explicitly writing that the public domain is allowed. There are a few things of note in this example: public values can be implicitly converted to private, the regular syntax for comparison can be used to invoke secure comparison, and conversion to the public domain requires an explicit call to **`declassify`** function. In order for Sharemind to be able to execute the following

Listing 3.5: Millionaires' problem in SecreC 2

```
import shared3p;
domain pd_shared3p shared3p;
void main () {
    pd_shared3p uint x = argument("IncomeAlice");
    pd_shared3p uint y = argument("IncomeBob");
    publish("result", x < y);
}
```

code, it has be configured to supply the mentioned security scheme and a concrete instance of it. The security schemes are implemented via loadable modules and concrete instances (so-called domains) have to be explicitly configured. To configure a protection domain a user has to specify the domains' name, which concrete servers are involved, and a loadable module that associates operation names with executable code. The precise details are very technical in nature, are subject to change and are thus not included in this work.

Input and output

In real applications private variables are rarely initialized directly from public ones like we just saw. A more realistic program would be one that, for example, gets the value of x from one client and y from another. As a result of executing the example both input providers would be notified whether x is less than y. This can be considered an implementation of the Millionaires' problem [113]. SecreC 2 code for such application is presented in Listing 3.5. Instead of declaring a protection domain kind we use appropriate standard library module that already declares that kind and defines various operators and functions for it. Inputs are no longer hard-coded and instead are given as arguments to the program. The result is published to output parties.

Notice that the example still only directly involves computing parties. It does not state from where the `"IncomeAlice"` and `"IncomeBob"` arguments come from or who receives the published result. This logic is implemented outside of SecreC 2. The very same code could be used in many different application. In one case it could be used for application that involves three parties two of which provide input and all of them receive the output. Or it could also involve a web interface for supplying data. In that case input, computing and result parties could all be distinct. For a real-world example of how SMC can be deployed for web applications see the dissertation [108] of Riivo Talviste.

Private conditionals

Private values can be used exactly like public ones in most places, such as, expressions, function arguments, variable declarations, and return values. However, they cannot be branched over. For example, if we want to conditionally change a secret value `x` depending on a private Boolean `b` we are not allow to write `if (b) { x = y;}`. The same result can be achieved by evaluating `(uint)b*(x-y)+y` that results in `x` if `b` is true and in `y` otherwise[1]. This pattern can be abstracted to a function.

Many SMC programming languages, for example, ObliVM [85] and PICCO [118], allow branching over secure Booleans in some restricted cases. We have decided to not adopt this approach main because we have not had the need for it. While it does help developers a little bit, in our experience development effort is not significantly increased in their absence. There are some other less important reasons for not having private conditionals.

1. When branching over secure Booleans, it is not possible to perform any public side effects such as printing output or performing public assignments. The type system or program analysis would need to detect such conditions. Encoding such information in types makes the language more complicated. On the other hand, achieving the same results via program analysis hides from the programmer if a function can be used in a private-conditional or not.

2. Secure branches are significantly slower than public ones because they force the compiler to evaluate both branches and then select the correct answer based on the private Boolean. This argument would be weak if private conditionals were indistinguishable from public ones, but they are not.

3. Supporting secure conditionals raises the question of why not also provide support for secure looping with an upper bound to the number of iterations.

In summary, supporting conditionals over secure Booleans adds value by helping the programmer but it also complicates the language, its type system, and compiler implementation. Having all control flow be public leads to a more approachable language and a simpler compiler. Given that private conditionals are not overly common we opted to not offer convenient syntax for them. This limitation is conservative in the sense that it can be relaxed in a backwards compatible manner if it turns out that private conditionals are frequently used.

[1]Depending on the security scheme, different expressions can achieve the same results more efficiently.

3.1.3 Protection domains and kinds

We provide two definitions that explain our motivation behind the choices for the keywords for declaring a security scheme and its instances.

Definition 5. A *protection domain kind* (PDK) is a set of data representations, algorithms and protocols for computing on and storing protected data.

Definition 6. A *protection domain* (PD) is a set of data that is protected with the same resources. There is a well-defined set of algorithms and protocols for computing on that data while keeping the protection.

Each protection domain belongs to one protection domain kind and each kind may have several protection domains. A classical example of a PDK is secret sharing (see Section 2.1.5) with an associated data representation and protocols for construction, reconstruction and arithmetic operations on shared values. A PD in this PDK would specify the actual parties doing the secret sharing. Another example of a PDK is fully homomorphic encryption (Section 2.1.3) with operations for encryption, decryption and algorithms for performing arithmetic operations such as addition and multiplication on encrypted values. For the FHE PDK, each PD is associated with a secret key. It is also possible to consider non-cryptographic methods for implementing PDK-s using trusted hardware or virtualization. Intel SGX offers an example of a practical PDK with a single physical computer evaluating a program.

In general a PDK has to provide:

1. A list of data types supported by the PDK.
2. For each data type in the PDK:
   (a) a *classification* function for converting public data to the protected representation and a *declassification* function for revealing private data,
   (b) protocols or functions for operating on protected values.

The protocols performing secure computation should be universally composable so that they can be safely combined into programs without losing security guarantees. Here we state that any PDK needs to allow its data to be converted to and from the public domain. This is not strictly necessary and a PDK may instead allow converting data to and from some other PDK. However, we have not found a practical case where restricting public conversions is useful.

Listing 3.6: A naive and slow implementation of counting in SecreC

```
pd_shared3p uint count (pd_shared3p uint32[[1]] data,
                        pd_shared3p uint32 value) {
    pd_shared3p uint matchCounter = 0;
    for (uint i = 0; i < size(data); ++ i)
        matchCounter += (uint) (data[i] == value);
    return matchCounter;
}
```

3.1.4 Array processing

Rarely do applications of secure computation deal with single values. More often the goal is to learn some information from a large set of data without revealing information about the individual values. Naturally, to process many records a programming language has to support arrays or some other forms of collections. As we will later see, a solid support for data-parallel processing is also an attractive feature. SecreC supports arbitrary-dimensional arrays with publicly known size. Access is provided and modifications are made via public indices.

A simple aggregation procedure is to count a number of occurrences of a value in a set of data. This can easily be achieved, as shown in Listing 3.6, by iterating over the input array, comparing with the value, and incrementing the counter whenever a match is found. Counter increments are achieved by casting the result of the equality check to an integer and adding that to the counter. When the comparison results in **true** the counter is incremented by 1 and otherwise by 0. Double square brackets, such as `[[1]]`, denote the array's dimensionality (the number of dimensions). For instance, public integer matrices are represented with `int[[2]]`. We use double square brackets to be distinct from C where `int[4]` denotes a one-dimensional array of length four. Stacking multiple square brackets as `int[][][][]` would mean that special syntax would be needed for dimensionality-generic code.

SecreC puts a lot of emphasis on array processing and tries to make it convenient. For example, many language constructs are lifted to operate pointwise on arrays. One can add two arrays using the exact same syntax as for adding two scalars. Binary arithmetic also allows for one operand to be an array and the other a scalar. Other than being less verbose vectorized operations are in many situations more efficient than loops of scalar operations. It is possible to rewrite the counting function in a more compact style by comparing the value to the data pointwise and summing the resulting Booleans as `sum(data == value)` where `sum` is a function defined in the standard library. We will later see that, in addition to conciseness, this version also performs significantly better than the one in Listing 3.6.

Reshaping and moving data between arrays is very common and to avoid writing verbose loops for such a typical task we support indexing arrays with ranges. Given a one-dimensional array `arr` the expression `arr[b:e]` results in an array of length `e-b` where the elements are taken from the range of indices from `b` to `e-1`. If omitted the lower bound is implicitly assumed to be 0 and the upper bound is assumed to be the length of the array. This syntax expands naturally to arbitrary-dimensional arrays and can be mixed with regular indexing. For example, the expression `mat[1,:]` results in the second column of the matrix `mat`.

It is possible to change the number of dimensions and dimension sizes of an array. The only requirement is that the number of elements of the resulting array be the same as in the input array. For example, a matrix `mat` can be flattened into a one-dimensional array with the `reshape(mat,size(mat))` expression. More generally, `reshape` takes arbitrary number of arguments after the first one that indicate the shape of the resulting array. SecreC guarantees that arrays are stored in generalized row-major order. When we flatten a matrix, its first row occurs first in the resulting array, then the second row, and so on. Scalars are allowed to be reshaped into any shape or size.

3.1.5 Protection domain polymorphism

Sharemind is not limited to using a single protection domain kind and can even support the use of multiple protection domains concurrently. This feature makes it possible to re-use code between different security schemes. For example, the function for counting occurrences in an array can be implemented for any scheme that supports comparison, addition, and conversion from Booleans to integers. However, the implementation in Listing 3.6 is not usable across different security schemes but only on one particular instance of the additive three-party scheme.

To enable reuse across security schemes, SecreC supports protection domain polymorphism using syntax similar to C++ templates. A generic version of the counting function is presented in Listing 3.7 where the function has been made polymorphic over any protection domain `D` but the rest of the code remains unchanged compared to the previous implementation. Note that this includes the public security domain, making the function also usable on public data. In many cases algorithms have a single implementation across multiple protection domains.

Template functions are type checked similarly to C++. Type correctness of a template function body is verified only when the function is instantiated to some concrete protection domain. The definition itself is only syntactically verified. Unfortunately this means that when the counting function is called on a protection domain that does not support equality, a type error is raised at the location of the comparison and not where the function is called from. Some other languages overcome a similar problem by restricting type parameters to only types that are known to offer some necessary functionality (such as equality). The restriction can

Listing 3.7: Protection domain generic counting function in SecreC

```
template <domain D>
D uint count (D uint32[[1]] data, D uint32 value) {
    D uint matchCounter = 0;
    for (uint i = 0; i < size(data); ++ i)
        matchCounter += (uint) (data[i] == value);
    return matchCounter;
}
```

be statically verified at the call site and error is raised when the constraints are not satisfied. For example, Haskell has type classes, rust has traits, and C++ has proposed concepts. All these solutions add considerable complexity to the language. Thus, we find it acceptable tradeoff to not have any form of bounded quantification. SecreC programs are at most medium-sized and typically span less than a few thousand lines of code.

Often it is not sufficient to provide a single implementation of a particular function that works on all protection domains. Sometimes we want a faster implementation for a certain protection domain, or the default implementation uses operations that a domain does not support. Function overloading can typically be used to solve those cases.

Consider function `choice` for choosing a value based on a Boolean. It takes a Boolean value and two arguments and returns the first argument if the Boolean is true and the second argument if the Boolean is false. SecreC does not support regular if-expressions over secure conditionals; thus, this kind of function is often useful for filling the role of branching. In Listing 3.8 the first definition is polymorphic over any protection domain but the second one is restricted to a domain based on Yao's garbled circuits scheme. The first definition can be considered a more generic default implementation and it relies on multiplication, addition and subtraction. However, for Yao's scheme multiplication can be a rather costly operation that we want to avoid. Instead, in the case of the `yao2p` domain, the Boolean is converted to a suitable bitmask and the result is computed via bitwise conjunction and XOR. Note that the second implementation is also not a good default as it is not efficient for additive secret sharing based schemes. The polymorphic overload that is specialized for `yao2p` protection domain kind works for any concrete domain of that kind.

When multiple matching definitions are found for the function being the most restricted match will be selected. In the current example, the second definition has a more restrictive signature than the first. When multiple equally matching functions are found, a type error is raised.

We have built a standard library using polymorphic functions, overloading and modules. The code is structured into a generic `stdlib` and specialized modules

Listing 3.8: Function overloading in SecreC

```
template <domain D>
D uint32 choice(D bool b, D uint32 x, D uint32 y) {
    return (uint32)b * (x - y) + y;
}

template <domain D : yao2p>
D uint32 choice(D bool b, D uint32 x, D uint32 y) {
    D uint32 mask = - (uint32) b;
    return mask & (x ^ y) ^ y;
}
```

Listing 3.9: Generic array flattening in SecreC

```
template <domain D, type T, dim N>
D T[[1]] flatten(D T[[N]] arr) {
    return reshape(arr, size(arr));
}
```

for each protection domain. The `stdlib` module defines functions that are polymorphic, with some overloaded on public domain. Each protection domain kind module can overload the standard library functions and provide the user with more efficient implementations. For example, we supply the `shared3p` module and some accompanying modules to provide more efficient operations.

In addition to protection domain polymorphism, we also allow functions to be generic over data types and array dimensionalities. For example, the function defined in Listing 3.9 reshapes an arbitrary-dimensional array of any type in any protection domain into a one-dimensional array.

3.2 Writing efficient SecreC

Efficiency is often a concern in secure computation. Secure algorithms, that are implemented directly based on some public version, can be infeasibly slow due to high round counts or impractical network bandwidth requirements. This is a problem as most secure computation protocols require network communication. Some security schemes require a constant number of communication rounds but high network bandwidth, and some schemes have low bandwidth requirements but need to perform many communication rounds. In either case, a programmer must take care when implementing secure algorithms.

In this section we provide some generic techniques that help improve performance of secure programs. We have the following suggestions that we will elaborate on and show how they are applied in SecreC.

1. Prefer parallel execution to sequential execution. Even for reasonably large inputs, applications should take a sublinear number of communication rounds.

2. In many cases the SIMD style of parallelism is sufficient but sometimes more involved techniques need to be applied to lower the number of communication rounds. For example, associative operations can be aggregated in a logarithmic number of rounds.

3. Prefer operations that are not costly in the chosen protection domain. For example, in additive schemes addition does not require network communication.

3.2.1 Parallel execution and SIMD

SecreC operates on the assumption that the underlying computation protocols are universally composable. This property allows secure operations to be executed either in parallel or sequentially without loss in security guarantees.

In real applications, when we execute two subsequent multiplications we invoke the underlying secure multiplication protocol twice sequentially. Regardless of how many communication rounds the underlying multiplication protocol performs, when two sequential multiplications are performed it takes about twice as much time. Compared to regular computers where operations take nanoseconds, in secure computation operations have latencies of over several milliseconds because of network communication. This is a difference in the order of 10^6.

Such a drastic overhead might sound like secure computing is completely infeasible. In addition to comparing only instruction latencies, we should also look at parallel execution. Namely, in a networked setting it is possible to perform thousands of operations in parallel in the same amount of time that it takes to execute a single operation. This is because the network is optimised for high-bandwidth usage. Thus, for reasonably efficient applications sequential execution should be avoided and parallel execution should be preferred.

SecreC has opted for *data parallelism* (performing one task on many pieces of data at the same time) over *task parallelism* (performing many tasks at the same time). Most operations in the language can be applied to arrays in which case the operation is executed pointwise on the data. This is also known as the *single instruction, multiple data* or *SIMD* approach.

To demonstrate the issue with high latency let us look at an implementation of dot product (Listing 3.10) as usually written in a regular imperative programming language. The common solution is to iterate over both arrays adding the product

Listing 3.10: Dot product in SecreC

```
template <domain D>
D uint32 dotProduct(D uint32[[1]] x, D uint32[[1]] y) {
    assert (size(x) == size(y));
    D uint32 result = 0;
    for (uint i = 0; i < size(x); ++ i)
        result += x[i] * y[i];
    return result;
}
```

Listing 3.11: Round-efficient dot product in SecreC

```
template <domain D>
D uint32 sum (D uint32[[1]] arr) {
    D uint32 result = 0;
    for (uint i = 0; i < size(arr); ++ i)
        result += arr[i];
    return arr;
}
template <domain D>
D uint32 dotProduct(D uint32[[1]] x, D uint32[[1]] y) {
    return sum (x * y);
}
```

of the corresponding elements to the accumulator. In most security schemes the multiplication operation requires the execution of a protocol that takes at least a single communication round. The iterative solution would then take a linear number of communication rounds with respect to input length. Due to network latencies this makes the algorithm infeasible to use in real applications with non-trivial input sizes.

To optimize the previous example we notice that we can multiply the input arrays pointwise. This is a data parallel operation and takes the same number of rounds as multiplication of scalar values. The result of the product is stored in a temporary array that is then summed. In the additive scheme, integer addition is a local operation; thus, the implementation in Listing 3.11 is efficient enough because iterative addition does not increase the round count of the procedure.

3.2.2 Round-efficiency

In some schemes, such as Boolean circuit based ones, addition is not a local operation, meaning that sequential addition still incurs a linear round count. This shortcoming can be overcome by iteratively performing data-parallel additions

Listing 3.12: Round-efficient summing in SecreC

```
template <domain D : pd_yao>
D uint32 sum (D uint32[[1]] arr) {
    uint n = size(arr);
    if (n == 0) return 0;
    while (n > 1) {
        uint smallHalf = n / 2; // Rounds down
        uint bigHalf = n - smallHalf;
        arr[: smallHalf] += arr[bigHalf : n];
        n = bigHalf;
    }
    return arr[0];
}
```

between the lower and upper half of the array until a singleton array remains. For an input of length n this procedure takes $\mathcal{O}(\log n)$ communication rounds. A possible implementation that also takes care of non-power-of-two inputs is provided in Listing 3.12. We could use round-efficient summation as the default implementation but it turns out to be slower in the case of public data and security schemes where addition is a local operation.

This technique is generic and is applicable to any associative binary operation. For example, secure floating-point addition as implemented by Kamm and Willemson [62] performs multiple communication rounds. Summing a sequence of floating-point values is a common operation that is often performance-critical. While floating-point addition is not strictly associative it is regardless very useful to apply this technique for performance reasons.

3.2.3 Security scheme specific techniques

Parallelism and round-reduction techniques are useful tools in general but especially important in a secure setting because of the very high constant factors involved. In many cases it is also possible to exploit the properties of the security scheme to provide even better performance. We shall see few of these techniques here and take a look how they can be implemented in SecreC.

Consider the task to check if at least one value in a Boolean array is true. An implementation that uses a parallel algorithm to compute the disjunction of all elements takes $\mathcal{O}(\log n)$ rounds for an n-element input array. In practice this is a perfectly reasonable implementation but it is possible to do better. Asymptotically improved round complexity is achieved by converting the Boolean array to additively shared integers, adding up the integers, and checking if the sum is zero. This exploits the fact that addition of additively shared integers is a local operation. In order to

Listing 3.13: Constant-round disjunction

```
template <domain D : shared3p>
D bool any (D bool[[1]] arr) {
    return sum ((uint64) arr) != 0;
}
```

not overflow the sum we must use at least $\lceil \log n \rceil$-bit integers. Because conversion from Boolean to integer takes a constant number of rounds and integer comparison has logarithmic round complexity, this yields an algorithm with $\mathcal{O}(\log \log n)$ round complexity. Our implementation in Listing 3.13 uses 64-bit integers which is sufficient for all practical purposes.

It is difficult to adopt many common algorithms to the secure computation setting. For example, most popular sorting algorithms branch their control flow depending on the results of comparisons between the elements of the input. These sorting algorithms cannot be adopted to secure computation setting in a straightforward manner because such control flow dependencies leak information, namely the ranks of the elements and, thus, the relative ordering of secure inputs. However, if the original ordering of the data is hidden by randomly shuffling (permuting) the input then the control flow does not reveal as much information. In fact, if all input values are unique then no information is revealed [51, 21].

A useful and efficient technique available for many different secret sharing schemes is to randomly shuffle the input to hide data dependencies. The random shuffling scheme proposed by Laur, Willemson and Zhang [80] is constant in round complexity and $\mathcal{O}(n \log n)$ in communication complexity. Thus, this primitive facilitates the implementation of $\mathcal{O}(n \log n)$ sorting algorithms. Many practical data-oblivious sorting algorithms exist—bitonic sorter [4], Batcher odd–even mergesort [4], pairwise sorting network [95]—but they all have $\mathcal{O}(n \log^2 n)$ time complexity, take $\mathcal{O}(\log^2 n)$ rounds, and are not stable. Numerous applications are built on top of oblivious sorting [74]. The shuffling primitive has also found use in oblivious database linking [79].

3.2.4 Trade-off between efficiency and privacy

When writing SMC code there is almost always a trade-off between efficiency and security. An implementation that does not reveal any information can be impractically slow. Revealing some sensitive information that is found to be acceptable to leak we can speed up the program significantly. One such example is sorting where the best practical oblivious algorithms take $\mathcal{O}(n \log^2 n)$ time. When it is acceptable to reveal the results of comparisons it is straightforward to adopt any $\mathcal{O}(n \log n)$ sorting algorithm to secure computation setting [21, 51]. We shall see how SecreC gives the programmer flexibility to make trade-offs between efficiency and privacy.

As a small case study we will implement *quickselect* [54] (also known as *Hoare's selection algorithm*) for finding the k-th smallest element of an unsorted sequence. On average the public algorithm runs in linear time. In this section we show how a naive secure implementation of the algorithm has $\mathcal{O}(n^2)$ running time but by revealing a little information about the input sequence we can achieve $\mathcal{O}(n \log n)$ average-case complexity, and by revealing even more we get an implementation that works in linear time.

Public quickselect operates similarly to quicksort. The algorithm operates iteratively on a working list. In every iteration we pick a pivot from the list and split the remaining list into two parts: elements smaller than the pivot, and the rest of the elements. Let the list wih smaller elements have l elements. If $k = l$ then we are done as the pivot is the k-th smallest element (we start counting from 0). If $k < l$ then the answer can be found in the list with smaller entries, and we just drop the larger elements and continue. If $k > l$ then the element can be found in the list with larger entries. In this case we drop the smaller elements and continue to look for the $(k - l - 1)$-th smallest element in the remaining sequence.

Usually, the development of a privacy-preserving algorithm starts from a public implementation or description. In Listing 3.14 we have implemented quickselect in SecreC so that it only operates on public data. It mostly follows the algorithm description from earlier but we have already given some forethought to the secure implementation. Namely, we perform all comparisons between the pivot and the rest of the list in parallel to keep the round count minimal. The results of the comparison are stored in the `mask` array and variable l is computed by summing the bitvector. If $k > l$ we flip the `mask`. After the checks we filter the list according to the mask. To keep the presentation simple, we have not optimized it. Namely, the algorithm does not operate in-place and traverses the sequence more often than is strictly needed.

Given an n-element input list, in the worst case our implementation may take n iterations leading to an $\mathcal{O}(n^2)$ algorithm. Consider the case where the list is already sorted and we are looking for the $(n - 1)$-th smallest element. In such case, during every iteration, the pivot is chosen to be the first element of the running sequence. Because it is the smallest one, no elements other than the pivot are eliminated from consideration. We can improve the algorithm by selecting the pivot randomly but this does not improve the worst case as every time the smallest element could be selected. A perfectly secure version has to account for the worst-case possibility. Thus, if we do not want to leak any information we have to perform all n iterations of the algorithm. But such an implementation is unacceptably slow and better ones clearly exist. For example, we could simply sort the input sequence using some oblivious sorting network and pick the k-th element of the result. That yields a

Listing 3.14: Public quickselect algorithm in SecreC

```
int quickselect(int[[1]] list, uint k) {
    assert (k < size(list));
    int pivot;
    while (true) {
        pivot = list[0]; // Random selection is better
        list = list[1:];
        bool[[1]] mask = list < pivot;
        uint l = sum(mask);
        if (k == l) break;
        if (k > l) mask = ~mask;
        if (k > l) k = k - l - 1;
        list = filter(list, mask);
    }
    return pivot;
}
```

much better $\mathcal{O}(n \log^2 n)$-time algorithm.

If we are willing to accept leaking the number of iterations that the algorithm performs, we can do much better. Unfortunately, this may actually give away a lot about the original data. For example, when we select the first element as the pivot during each iteration and the algorithm performs n iterations then we learn that the input list is sorted. Randomized pivot selection makes such a leak unlikely but does not eliminate it.

The secure implementation in Listing 3.15 follows quite straightforwardly from the public one. To stop iterating, we need to declassify the results of comparison $k = l$. We have to update variables `mask` and `k` obliviously. Updating the working list, however, is tricky. Namely, we cannot simply obliviously remove unnecessary elements as this will give away more information than we would like. One possible solution is to use an additional bit vector indicating if an element in the list is actually there or not. This would complicate the implementation. A simpler solution is to replace elements that are not needed with some reserved values that are guaranteed to be larger than all others, and move them to the end of the sequence.

Secure partitioning is performed in the following way. First we replace the elements of the sequence corresponding to false values in the mask by maximum integer values, and then we sort the sequence by the Boolean mask moving true values to the front. This can be done similarly to the Boolean counting sort step of the radix sort in [108, Algorithm 8]. Sorting Boolean vectors can be done in effectively linear time and; hence, the partitioning is also linear-time and takes a constant number of communication rounds. Given that, we can say that on average the algorithm in Listing 3.15 takes $\mathcal{O}(n \log n)$ time and $\mathcal{O}(\log n)$ communication

Listing 3.15: Privacy-preserving $\mathcal{O}(n \log n)$ quickselect algorithm in SecreC.

```
template <domain D>
D int quickselectPrivate(D int[[1]] list, uint k) {
    assert (k < size(list));
    D int pivot;
    while (true) {
        pivot = list[0];
        list = list[1:];
        D bool[[1]] mask = list < pivot;
        D uint l = sum(mask);
        if (declassify(k == l)) break;
        mask = choose(k > l, ~mask, mask);
        k = choose(k > l, k - l - 1, k);
        list = partition(list, mask);
    }
    return pivot;
}
```

rounds. This is already as good as sorting-based solutions.

To go even further, if we are willing to leak on every iteration the number of elements smaller than the pivot, we can achieve linear-time solution. When declassifying l we can, instead of partitioning, remove the unneeded elements; thus, reducing the size of the working list. To improve security we can pick the pivot securely. This can be done by securely shuffling the working list at the start of every iteration and continuing as before.

3.3 Compiler implementation and integration with SHAREMIND

In this section we give a short overview of the SecreC compiler implementation and its integration with the Sharemind framework. We will also briefly discuss the bytecode to which the language compiles.

3.3.1 Compiler implementation

SecreC is implemented in the C++ programming language and the source code is available[2] under the GPLv3 licence. The compiler follows the standard compilation pipeline: lexical analysis, syntactic analysis, semantic analysis (type checking), intermediate code generation and, finally, code generation. We target a custom

[2]https://github.com/sharemind-sdk/secrec

bytecode that Sharemind is able to execute. Lexical analysis was implemented using Flex and syntactic analysis using GNU Bison.

3.3.2 Custom bytecode

Sharemind is able to execute custom bytecode. We will not go over the details of the bytecode but we will elaborate on why a custom bytecode language is used. The bytecode specification is available online[3].

We use custom bytecode for a few reasons. The most important one is security. By sacrificing some performance, we can have a more secure implementation where invalid operations, like division by zero, can gracefully stop the execution of the code instead of crashing the entire platform. Many bytecode languages have access to foreign functions and raw memory but this is something we wish to avoid for the sake of security. Custom bytecode allows for fine-grained control of memory. A program can be stopped by the VM when it goes over some memory allocation limit. Because we avoid garbage collection, the programmer can better predict memory usage.

The secondary reason for having custom bytecode is portability. Namely, different architectures implement floating-point operations in a different manner. For example, machine instructions for computing sine might give one result on one machine but a slightly different result on another. In such case the control flow of a program executing on these machines could dffer. Many existing bytecode languages rely on floating-point operations and therefore not suitable for distributed execution necessary for SMC. Sharemind's bytecode relies on a software implementation of floating-point operations. Currently, we use the SoftFloat [52] (Release 2c) library.

3.3.3 Integration with Sharemind

The SecreC language is not tightly coupled to Sharemind. The compiler is just a standalone tool that takes a SecreC program text and produces a bytecode executable. The bytecode, in turn, can be executed using Sharemind. Conceptually, the bytecode can be generated from some other language or can even be manually written.

The SecreC compiler does not have to be deployed together with Sharemind but it usually is. This is because Sharemind is still vulnerable to executing untrusted code and, therefore, before executing any code it must be reviewed. Bytecode is much more difficult to audit than high-level SecreC programs. Hence, system administrators usually compile the SecreC file themselves after auditing the code, and then manually deploy the compiled bytecode file. The SecreC compiler is open-sourced and free to be verified by anyone.

[3]https://github.com/sharemind-sdk/vm_m4/blob/master/doc/bytecode.md

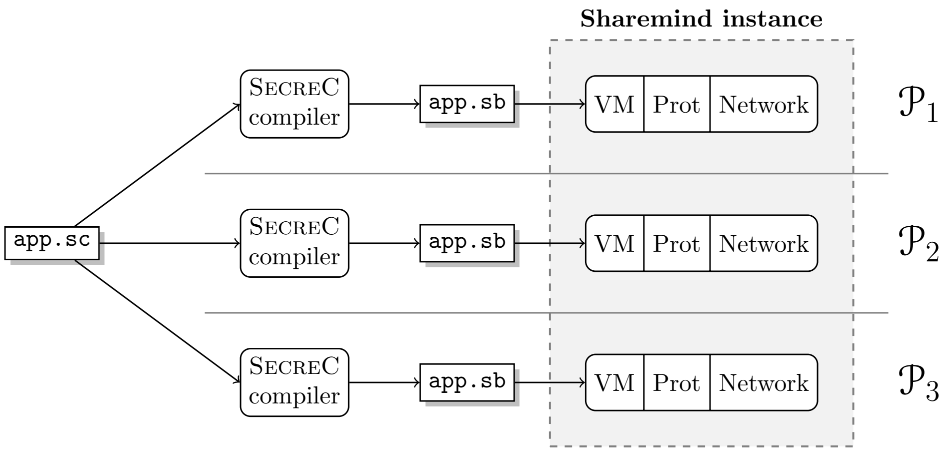


Figure 3.1: SecreC integration with a Sharemind instance running on three parties $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$

A standard Sharemind deployment scheme is given in Figure 3.1. The figure depicts how a SecreC script `app.sc` is compiled and run on a Sharemind instance. For each of the three parties, the script is compiled to a bytecode file `app.sb`. When instructed, the Sharemind instance can then execute the deployed script. Before execution, Sharemind checks that the bytecode files are identical.

The bytecode contains the list of protection domains that it expects to use, and also the list of system calls that it may perform. Before execution, Sharemind verifies if such protection domains have been configured and checks that all system calls are supported. After verification, Sharemind instantiates the required protection domains. This can include setting up various facilities such as random number generators, local overlay networks, database engines, and logging back-ends. If everything goes well, the program is executed after the setup phase.

The individual bytecode instructions do not include network communication, random number generation or any other security primitives. All of these tasks are performed via system calls to Sharemind. When the bytecode invokes a system call, control is given to the protection domain that implements it. Generally, a system call first verifies that its parameters (e.g., number of arguments, input lengths, input types) are correct, and then executes the actual logic. For example, the system call could perform the additive three-party multiplication protocol. To execute that protocol, protection domains need to generate random numbers and perform network communication between each other.

3.4 Future language extensions

SecreC is already a very usable language with many applications and prototypes that utilize it but much work remains to be done to improve the language. SecreC is currently quite simple, makes too many assumptions about underlying security schemes, lacks many optimizations and offers little protection from simple mistakes concerning security. We will explore a few possible and upcoming language extensions that can alleviate these problems.

3.4.1 Specifying protection domains

Currently the most critical limitation of the language is that it assumes the existence of a large set of primitives from the underlying protection domains. For example, when a user attempts to divide two unsigned 64-bit integers, SecreC assumes that the security domain supports a call with the name `div_uint64` and the compiler generates code where such a system call is performed. If the underlying domain does not support division then a run-time error will be raised when trying to execute the program. The programmer does not learn that the operation is not supported until the code is actually ran. While emulation of protection domains can simplify development to some degree, this is still less than ideal situation.

We also assume that protection domains support all data types that SecreC has. This might not be true, for example, a basic experimental scheme could only support 32-bit integers. Another outstanding problem is that currently SecreC does not allow protection domains to have types that do not exist publicly. It might be that a security domain only support fields over some prime numbers. In that case the data type should be different from all the public types.

Sokk [106] extended the SecreC language with support for specifying the interface of protection domain kinds in the language itself. As a result of his work it is now possible to define custom data types for domains and describe which operations are available for the given types. For example, we can declare a new protection domain that supports 64-bit signed integers, fixed-point numbers, and 64-bit unsigned integers as is shown in Listing 3.16. For each type the programmer must specify the size of the private representation and declare its public representation. Both the size and the public representation can be omitted.

3.4.2 Other language extensions

SecreC is currently a simple language. There are many features that can be borrowed from general-purpose high-level languages that would make developers' lives easier. For example, the language only supports passing function arguments by value. While general first-class pointers or references would arguably not be

Listing 3.16: Specifying PDK interface in SecreC.

```
kind my_kind {
    type int64 { public = int64, size = 64 };
    type fix { public = float64, size = 64 };
    type uint64;
}
```

suitable for a secure computing language we would benefit a lot from allowing function arguments to be passed by reference. A few more possible additions are first class support for structures and strings, more powerful abstraction mechanisms like template specialization and type constraints, language support for task-level parallelism (even parallel loops) and more syntactic sugar for array processing.

The type system of SecreC can catch simple coding errors and accidental implicit declassifications. However, it is still possible to accidentally reveal more information than what was intended via explicit declassification calls. A more powerful type system or static analysis tools will help in this regard. Some work has already been done to verify SecreC annotations for both correctness and security guarantees. We give a short overview of this tool in Section 6.7.

3.4.3 Optimizations

Currently SecreC does not perform any code optimizations. However, we could benefit from both optimizations described in general compiler literature and optimizations specific to secure computation For example, Kerschbaum [65] describes an optimization that infers if any of the secure computations can be made public. This is possible if the values of intermediate secure results can be inferred from public inputs or outputs. The *knowledge inference* problem was later expanded upon by Rastogi [101].

CHAPTER 4

THE SEMANTICS OF SECREC

In order to talk about correctness and security properties of SecreC programs, we need to specify the language formally. A formal specification can also be useful for understanding of the language better and can reflect some details more clearly than informal specification. We do not formalize the language fully and limit ourselves to a rather small core language. There are a few reasons why we have opted not to fully formalize the language.

- For a small development team with different focus a total specification constitutes an impractical amount of work. In our case this time is better spent improving the language.
- Just like documentation and comments, a formal specification is easily subject to bit rot and needs to be kept in sync with the development of the language.
- Formal specification is highly likely to contain errors.
- Many language features, such as structures, are completely orthogonal to the properties that we wish to prove and, thus, specifying them provides us with little value.

Using the formal approach, we wish to show that if the underlying computing protocols are secure and compose well then programs built of only those protocols are also secure. Additionally, we wish to show that translating a program to a low-level representation preserves the security of the original program. We limit our scope to only monomorphization, that is, turning a program that contains security scheme polymorphic functions into one that only contains monomorphic functions. This represents compilation to a lower-level representation and is one of the important steps in the code generation pass of the compiler.

In this chapter we provide the formal foundations of the language. This includes the formalization of the syntax and semantics of the core language, a formal

$$
\begin{array}{rcl}
P & ::= & (\mathsf{pdk}\ k \mid \mathsf{pd}\ d : k \mid F)^*\ s \\
F & ::= & [\forall \alpha_1, \ldots, \alpha_m]\ f(x_1 : d_1\ t_1, \ldots, x_n : d_n\ t_n) : d\ t\ [s] \\
\alpha & ::= & d \mid d : k \\
t & ::= & \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{int[]} \mid \texttt{bool[]}
\end{array}
$$

$$
\begin{array}{rcl}
s & ::= & \texttt{skip} \\
 & \mid & s_1;\ s_2 \\
 & \mid & x = e \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\
 & \mid & \texttt{while}\ e\ \texttt{do}\ s \\
 & \mid & \texttt{return}\ e \\
 & \mid & \{x : d\ t;\ s\}
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & x \\
 & \mid & c_t \\
 & \mid & e_1 + e_2 \\
 & \mid & f(e_1, e_2, \ldots, e_n)
\end{array}
$$

Figure 4.1: The syntax of the SecreC core language

description of translation to a monomorphic form and a discussion about the correctness of translation.

4.1 Abstract syntax

The abstract syntax of SecreC core is described in Figure 4.1. Programs are denoted with P and consist of a sequence of protection domain kind declarations, protection domain declarations, and function definitions, followed by the program body. A program body consists of a single statement.

We denote variables with x, kind names with k, function names with f, and protection domain names with d. Protection domain `public` and kind `Public` that it belongs to, are reserved and considered predefined. We sometimes write public types, such as `public int`, as $\texttt{int}^+$ for the sake of conciseness.

A function definitions F consist of a function name f, a sequence of parameter declarations, a return type, and finally a function body. The function body may be missing in which case the declaration is considered to be a system call provided by Sharemind. This is similar to the foreign function call mechanism in general-purpose programming languages. Functions in the SecreC core may be protection domain polymorphic and in this case the function needs to be annotated with the universal quantifier followed by some protection domain declarations α. A universally quantified domain may optionally be restricted to some protection domain kind. This is useful for security scheme specific operations and polymorphic function overloading to offer more efficient operations for some security schemes.

Data types in core SecreC have been limited to the unit type (needed for functions that do not return anything), integer type, Boolean type, and arrays of integers and booleans. Data types are denoted with t.

Statements s and expressions e (Figure 4.1) are fairly standard for a WHILE-language. Expressions only contain variables, literals, function calls, and addition. Other binary operators, such as multiplication, are very similar to addition where semantics and type checking are concerned and can always be emulated via either function or system calls. We assume that for public literals c_t it is always possible to tell their data type t. Statements s are limited to skip, composition, assignment, if, while, and return statements, and variable declarations.

4.2 Notation

In order to specify the static and dynamic semantics we need to define some notation.

- As SecreC allows function overloading we need a way to distinguish different function definitions that have a common name. Let $\mathcal{L}$ be a set of labels such that every function declaration has a unique label from this set. For label $\ell \in \mathcal{L}$ let f^ℓ denote the function (or system call) declaration F with the name f at location ℓ.

- Let $\mathsf{pdk}(P)$ denote the set of protection domain kinds that are declared in program P. Similarly, let $\mathsf{pd}(P)$ denote the set of protection domains declared in P. For $d \in \mathsf{pd}(P)$, let $\mathsf{kind}(d) \in \mathsf{pdk}(P)$ denote the declared kind of the protection domain d.

- For a function declaration f^ℓ in program P let $\mathsf{arg}_P(\ell)$ denote the list of declarations of function's formal parameters and $\mathsf{ret}_P(\ell)$ its return type.

- For a label $\ell \in \mathcal{L}$ let $\delta(\ell)$ denote the set of protection domain names that the function declaration f^ℓ is quantified over. If the function declaration f^ℓ is not a template then this set is empty.

- Let $\mathsf{body}_P(\ell)$ denote the body of the function declaration at the location $\ell \in \mathcal{L}$. If the function declaration at ℓ refers to a system call then let $\mathsf{body}_P(\ell)$ be $\perp$. Let $\mathsf{body}(P)$ denote the body of the program P.

- Let $\theta : \delta(\ell) \to \mathsf{pd}(P)$ denote a substitution from quantified variables of f^ℓ to protection domains of the program P. Given a mapping θ, let $\mathsf{body}^\theta_P(\ell)$ denote the statement $\mathsf{body}_P(\ell)$ where all occurrences of quantified variables $d \in \delta(\ell)$ have been syntactically replaced by protection domains $\theta(d)$. We define $\mathsf{arg}^\theta_P(\ell)$ and $\mathsf{ret}^\theta_P(\ell)$ in a similar manner by replacing all quantified variables of f^ℓ using θ.

- For a function name f and program P let

  $$\mathsf{instance}_P(f; d_1\ t_1, \ldots, d_n\ t_n) : [\mathcal{L} \times (\delta(\ell) \to \mathsf{pd}(P))] \cup \{\bot\}$$

  denote a pair where the first component is a label ℓ and the second component is a substitution θ from the quantifiers of f^ℓ to protection domains of P. The function f^ℓ is the best match among n-ary function declarations named f with parameters that under substitution θ are equal to respective $d_i\ t_i$. If a unique best matching instance is not found the resulting value is $\bot$.

The $\mathsf{instance}_P(f; \cdot)$ function is responsible for finding both the best matching function declarations f and the substitution that makes the function's type equal to the provided argument types. Note that the body of the function found may not type check under the given substitution. In this case the type checker later fails when verifying the function's instantiation to concrete protection domains.

A matching function may be missing for two reasons, either because there are no matches or because there are multiple equally good ones. We do not specify the way matches are evaluated against each other. One way to do that is to build subtyping relations between function signatures, then consider only the subset of all signatures that match and check if a unique least element is found among them. The least element in this case means the most concrete signature. For example, monomorphic signatures that match are always strictly better than polymorphic ones.

4.3 Static semantics

Let I denote a set of triples (f, ℓ, θ) where f is a function name, $\ell \in \mathcal{L}$ is a location and $\theta : \delta(\ell) \to \mathsf{pd}(P)$ is a substitution from quantifiers of f^ℓ to protection domains of program P. Intuitively, set I contains all template instances that are required by the program. We say that the program P is well-typed if there exists a finite instantiation context I such that $I \vdash P$. Each instance in I itself needs to be well-typed. Whenever a function is called the respective instance needs to be in I. Before we look at type checking rules formally, we also need to consider the following typing judgements.

- Let $P; \theta; I \vdash f^\ell$ denote that the function definition or system call declaration with name f at location ℓ is well-typed in program P if the quantified variables $\delta(\ell)$ of f^ℓ have been replaced using substitution θ.

- Let $P; \Gamma; I \vdash e : d\,t$ denote that expression e has a type $d\,t$ in program P under type environment Γ. The environment Γ is of the form $x_1 : d_1\ t_1, \ldots, x_n : d_n\ t_n$ where x_i are variables and $d_i\ t_i$ are types of the variables. For an

$$\frac{P; \emptyset; I \vdash \mathsf{body}(P) \quad \forall (f, \ell, \theta) \in I.\ P; \theta; I \vdash f^{\ell}}{I \vdash P} \ (\textsc{Program})$$

Figure 4.2: Program type checking rule

$$\frac{\mathsf{body}^{\theta}_{P}(\ell) = \bot}{P; \theta; I \vdash f^{\ell}} \ (\textsc{SysCall})$$

$$\frac{\mathsf{body}^{\theta}_{P}(\ell) \neq \bot \quad P; (\mathtt{return} : \mathsf{ret}^{\theta}_{P}(\ell), \mathsf{arg}^{\theta}_{P}(\ell)); I \vdash \mathsf{body}^{\theta}_{P}(\ell)}{P; \theta; I \vdash f^{\ell}} \ (\textsc{Function})$$

Figure 4.3: Function declaration type checking rules

expression e to be considered well-typed, all its free variables have to have a type in Γ.

- Finally, let $P; \Gamma; I \vdash s$ denote that statement s is well-typed in program P under type environment Γ.

A program P is well-typed if its body is well-typed under the empty type environment and all instances in I are also well-typed. This is reflected by the derivation rule in Figure 4.2. Notice that for a program to be well-typed, it is sufficient for only the instances that are actually used to be well-typed. Therefore, a well-typed program may contain procedure definitions that would not be well-typed, as long as these functions are never actually used.

Figure 4.3 presents typing derivations for system call declarations and procedure definitions. System call declarations are always considered well-typed. However, a procedure definition f^{ℓ} is well-typed under a substitution θ if its body $\mathsf{body}^{\theta}_{P}(\ell)$, substituted by θ, is well-typed in the environment where procedure parameters have types given by $\mathsf{arg}^{\theta}_{P}(\ell)$. The type checking environment also contains a special variable `return` that denotes the return type of the function instance, needed for type checking return statements.

Expression type checking rules are presented in Figure 4.4. Variable x has type $d\ t$ if it has that type in the environment Γ. A constant c of type t always has a public type. Recall that we assume that it is always possible to tell the data type of a constant literal. The sum of expressions e_1 and e_2 has type $d\ t$ if both expressions have type $d\ t$ and t is either an integer or an integer array. Because we allow addition of both public and private integers we have made an implicit assumption that every protection domain kind supports addition. Of course, this is not always so and it is sensible to allow user-defined non-public operators for convenience and type safety.

$$\frac{(x : d\,t) \in \Gamma}{P; \Gamma; I \vdash x : d\,t}\ (\textsc{Var}) \qquad \frac{}{P; \Gamma; I \vdash c_t : t^+}\ (\textsc{Lit})$$

$$\frac{P; \Gamma; I \vdash e_1 : d\,t \quad P; \Gamma; I \vdash e_2 : d\,t \quad t \in \{\texttt{int}, \texttt{int[]}\}}{P; \Gamma; I \vdash e_1 + e_2 : d\,t}\ (\textsc{Add})$$

$$\frac{\begin{array}{c} P; \Gamma; I \vdash e_1 : d_1\,t_1 \quad \ldots \quad P; \Gamma; I \vdash e_n : d_n\,t_n \\ (\ell, \theta) = \mathsf{instance}_P(f; d_1\,t_1, \ldots, d_n\,t_n) \quad (f, \ell, \theta) \in I \end{array}}{P; \Gamma; I \vdash f(e_1, \ldots, e_n) : \mathsf{ret}^{\theta}_P(\ell)}\ (\textsc{Call})$$

Figure 4.4: Expression type checking rules

Finally, a function call is well-typed if the function arguments are well-typed, there exists a unique instantiation to the argument types, and that instance can be found in I.

Statement type checking rules are presented in Figure 4.5. Most of the rules are straightforward and are easy to adopt from standard WHILE-language type checking. Two notable rules are for if- and while-statements. Namely, in both cases the condition has to be a public Boolean. This is because control flow is public in SecreC and selecting the branch based on a private Boolean will leak that secret value. It is possible to support certain classes of if-statements with private conditionals but it would complicate the type system. For example, secure branches may not perform public side effects but may perform private side effects like assignments to private values. To spare the programmer worries about such complexities we have decided that non-public branching always has to be explicit.

In our previous work [20] we presented the type checking rules without needing instantiation context but we required that the derivation rules be interpreted co-inductively. The co-inductive rules are indeed simpler but unfortunately make the interaction between inductively defined translation rules more complicated.

There may exist many different instantiation contexts for which a program is well-typed. We are usually only interested in the smallest one. Let I be an instantiation context and P a program such that $I \vdash P$. Let G be a directed graph where vertices are the instances from I and let arcs transition from instance i to instance j if the instance j is required to type check the instance i. Let G also contain a special node `src` that denotes the body of program P. We have an arc from vertex `src` to vertex i if i is required to type check `src`. To type check program P, only instances that are reachable from `src` are required. Unreachable instances can be thrown away. Instances reachable from `src` cannot be discarded as they are (indirectly) needed to type check the body of program P. This discussion gives us the following result.

$$\frac{}{P;\Gamma;I \vdash \mathsf{skip}}\ (\textsc{Skip}) \qquad \frac{P;\Gamma;I \vdash s_1 \quad P;\Gamma;I \vdash s_2}{P;\Gamma;I \vdash s_1;\ s_2}\ (\textsc{Cons})$$

$$\frac{d \in \mathsf{pd}(P) \quad \Gamma' = (\Gamma, x : d\,t) \quad P;\Gamma';I \vdash s}{P;\Gamma;I \vdash \{x : d\,t;\ s\}}\ (\textsc{Decl})$$

$$\frac{(x : d\,t) \in \Gamma \quad P;\Gamma;I \vdash e : d\,t}{P;\Gamma;I \vdash x = e}\ (\textsc{Assign})$$

$$\frac{(\mathtt{return} : d\,t) \in \Gamma \quad P;\Gamma;I \vdash e : d\,t}{P;\Gamma;I \vdash \mathsf{return}\ e}\ (\textsc{Return})$$

$$\frac{P;\Gamma;I \vdash e : \mathsf{bool}^{+} \quad P;\Gamma;I \vdash s_1 \quad P;\Gamma;I \vdash s_2}{P;\Gamma;I \vdash \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2}\ (\textsc{If})$$

$$\frac{P;\Gamma;I \vdash e : \mathsf{bool}^{+} \quad P;\Gamma;I \vdash s}{P;\Gamma;I \vdash \mathsf{while}\ e\ \mathsf{do}\ s}\ (\textsc{While})$$

Figure 4.5: Statement type checking rules

Proposition 1. *For a program P if there exist I_1 and I_2 such that $I_1 \vdash P$ and $I_2 \vdash P$ then $I_1 \cap I_2 \vdash P$.*

Type checking algorithm A program is defined to be well-typed through the existence of a certain instantiation context. Thus, these rules do not directly give a type checking algorithm. Fortunately, the type checking algorithm is quite straightforward to implement. We need to keep track of the instances that have already been type checked and keep a queue of instances that still remain to be checked. Initially, no instance is checked and the queue is empty. A new instance is added to the queue whenever a function call is type checked. We start the algorithm by checking the body of the program. We take and remove instances from the queue until there are no more instances to check. If the instance is already checked we pick another one. Otherwise, we check the instance and mark it checked.

The type checking algorithm clearly terminates because a program may only declare a finite number of protection domains. If we allowed procedures to be generic over types or the number of array dimensions we would need some extra conditions to guarantee termination of type checking.

4.4 Dynamic semantics

In this section we describe the behavior of SecreC programs in small-step operational style. We chose the small-step style as it allows us to reason about non-terminating programs and we found it a more natural fit than big-step or denotational style.

Let $\mathbf{Val}$ denote the set of values SecreC expressions may take. We have left the structure of $\mathbf{Val}$ open but it definitely contains Booleans (`true` and `false`), integers, Boolean arrays and integer arrays. A special value $\bot \in \mathbf{Val}$ denotes uninitialized or undefined values. Let $\mathbf{PolyVal}$ contain triples (v, d, t), usually written $v_{d\ t}$, denoting that the value $v \in \mathbf{Val}$ is in protection domain d and has data type t. Dynamic tracking of types is only needed for the polymorphic language semantics. Later we will see how to convert programs to monomorphic form in which we do not need to dynamically keep track of types. We usually denote values with v but sometimes also with u and w. To provide the polymorphic language with dynamic semantics we need to extend the set of expressions e with the set of values: $e ::= \ldots \mid v_{d\ t}$ where $v_{d\ t} \in \mathbf{PolyVal}$.

4.4.1 Evaluation context

An *expression evaluation context* $\mathcal{E}$ is an expression where exactly one subexpression is replaced with a hole denoted with "$\bullet$". We define $\mathcal{E}$ so that only the leftmost unevaluated subexpression can be replaced with a hole. This means that all subexpressions to the left of the $\bullet$ have to be fully evaluated to values. We define $\mathcal{E}$ as follows:

$$\begin{array}{rcl} \mathcal{E} & ::= & \mathcal{E} + e_2 \\ & \mid & v_{d\ t} + \mathcal{E} \\ & \mid & f(v^1_{d_1\ t_1}, \ldots, v^{i-1}_{d_{i-1}\ t_{i-1}}, \mathcal{E}, e_{i+1}, \ldots, e_n) \\ & \mid & \bullet\ . \end{array} \tag{4.1}$$

The evaluation context is used to specify the order of evaluation of subexpressions. Our definition effectively fixed evaluation order to be from left to right. In principle, there are other options. For instance, the C language does not specify the order of evaluation at all. It might sound like an unintuitive and easily misused aspect but it allows for C programs to be more aggressively optimized. Regardless, we chose a fixed evaluation order as it is easier to understand and (formally) reason about. The efficiency of public operations is not our primary concern.

We define *statement evaluation context* $\mathcal{S}$ in a similar manner. A statement evaluation context $\mathcal{S}$ contains a single expression evaluation context which in turn contains a single hole for an expression.

$$\begin{array}{rcl} \mathcal{S} & ::= & \mathcal{S};\, s_2 \\ & | & x = \mathcal{E} \\ & | & \texttt{return}\, \mathcal{E} \\ & | & \texttt{if}\, \mathcal{E}\, \texttt{then}\, s_1\, \texttt{else}\, s_2\ . \end{array} \tag{4.2}$$

Notice that while-statements are not represented in $\mathcal{S}$. As we will later see this is because the evaluation of while-statements is done via rewriting to if-statements.

Let $\mathcal{E}[e]$ denote an expression where the single hole in $\mathcal{E}$ has been replaced by expression e. This operation is defined via structural recursion over $\mathcal{E}$.

$$\begin{array}{rcl} (\mathcal{E} + e_2)[e] & = & \mathcal{E}[e] + e_2 \\ (v_{d\,t} + \mathcal{E})[e] & = & v_{d\,t} + \mathcal{E}[e] \\ f(v^1_{d_1\,t_1}, \ldots, \mathcal{E}, e_i, \ldots, e_n)[e] & = & f(v^1_{d_1\,t_1}, \ldots, \mathcal{E}[e], e_i, \ldots, e_n) \\ \bullet[e] & = & e \end{array}$$

In some cases we will use $\mathcal{E}[e]$ in argument position. This indicates an expression e' with the leftmost unevaluated expression e such that $e' = \mathcal{E}[e]$. It is always unambiguously clear if for a given expression e' there exists $\mathcal{E}$ such that $e' = \mathcal{E}[e]$ either for any or for some fixed e. Statement $\mathcal{S}[e]$ is defined in a similar manner.

4.4.2 Small-step operation semantics

During program evaluation we store the values of variables in an *environment* γ. We represent γ as a partial mapping from variables to values. For a variable x, let $\gamma(x) \in \mathbf{PolyVal}$ denote its value in γ. Given a value $v_{d\,t} \in \mathbf{PolyVal}$, a variable x and an environment γ let $\gamma[x \mapsto v_{d_t}]$ denote environment γ' such that $\gamma'(y) = v_{d\,t}$ if $y = x$ and $\gamma'(y) = \gamma(y)$ otherwise.

When evaluating the body of a function we need to keep track of the values of its local variables. When a function is called we need to store the local environment γ and remember the current position of evaluation. The context where the function call was performed is represented using statement evaluation context $\mathcal{S}$. A *configuration* C is a non-empty sequence $C ::= \gamma \mid (\mathcal{S}, \gamma) : C$ that keeps track of the call stack and the current evaluation environment. All components of C, other than the last one, are pairs consisting of a statement evaluation context $\mathcal{S}_i$ and an environment γ_i. The last component is simply the environment that is currently being used for evaluating either the program body or the body of some function.

The semantics of SecreC are given in small-step operational style as a set of tuples $P \vdash C^\circ \xrightarrow{\kappa} C^\bullet$ where C° is called *program configuration*, $C^\bullet$ is called *target configuration* and $\kappa \in \mathcal{A}$ is the *action* performed during the transition from one configuration to the next. The action κ can either be empty (denoted with τ or simply omitted) or an invocation of a system call.

$$\frac{v_{d\ t} = \gamma(x)}{\gamma \vdash \langle x \rangle \Rightarrow \langle v_{d\ t} \rangle} \quad \frac{d = \texttt{public} \quad v = c_t}{\gamma \vdash \langle c_t \rangle \Rightarrow \langle v_{d\ t} \rangle} \quad \frac{d = \texttt{public} \quad w = u + v}{\gamma \vdash \langle u_{d\ t} + v_{d\ t} \rangle \Rightarrow \langle w_{d\ t} \rangle}$$

$$\frac{d \neq \texttt{public} \quad k = \mathsf{kind}(d) \quad \kappa \equiv (w = \texttt{add}_{k,t}(d, u, v))}{\gamma \vdash \langle u_{d\ t} + v_{d\ t} \rangle \overset{\kappa}{\Rightarrow} \langle w_{d\ t} \rangle}$$

Figure 4.6: Expression evaluation rules

System call actions are of the form $v = f(d, d_1, \ldots, d_n, v_1, \ldots, v_n)$ denoting that the n-ary system call with name f, given parameters $v_1, \ldots, v_n \in \mathbf{Val}$ such that v_i is from domain d_i, returns value $v \in \mathbf{Val}$. The domain d for the return value v is also given as an argument. We assume that given the name of the system call it is possible tell both the kind and data type of its arguments and the result. In other words, the system call with name f must have a signature $\prod_{i=1}^{n}(k_i, t_i) \rightarrow (k, t)$ such that $k_i = \mathsf{kind}(d_i)$, $k = \mathsf{kind}(d)$, t_i are the data types of the arguments and t is the data type of the result. This information must be encoded in the name of the system call action. We do not pass to system calls the values in $\mathbf{PolyVal}$ as we want the set of actions $\mathcal{A}$ to be compatible between the monomorphic and the polymorphic language.

A program configuration C° is either a program $\langle P \rangle$ or a pair $\langle C, s \rangle$ consisting of a configuration and a statement s that is being evaluated. The target configuration $C^\bullet$ is either a configuration C when evaluation has halted, or a pair $\langle C, s \rangle$ when the evaluation of statement s needs to be continued.

Let tuple $\gamma \vdash \langle e \rangle \overset{\kappa}{\Rightarrow} \langle v_{d\ t} \rangle$ denote that expression e evaluates to $v_{d\ t} \in \mathbf{PolyVal}$ in the environment γ. Expression evaluation rules are presented in Figure 4.6. These rules only handle the evaluation of constants, variables and additive expressions with fully evaluated sub-expressions. Constants are mapped directly to values, values of variables are found in environment and public additions are performed without invoking any actions. Function calls are part of expressions but they are actually handled by the statement evaluation rules.

When non-public integers are added we perform a system call action with name $\texttt{add}_{k,t}$ where k is the protection domain kind of the arguments and t is the data type of the arguments. The system call is passed and returns monomorphic values and, therefore, the system call is incapable of telling the types of its arguments. Thus, type information needs to be encoded in names of system calls. In the case of arithmetic operations it is sufficient to only supply one kind and one type because they are the same for the arguments and the result.

$$\frac{P \vdash \langle C, s\rangle \xrightarrow{\kappa} \langle C', s'\rangle}{P \vdash \langle (\mathcal{S}, \gamma) : C, s\rangle \xrightarrow{\kappa} \langle (\mathcal{S}, \gamma) : C', s'\rangle} \quad \frac{P \vdash \langle C, s\rangle \xrightarrow{\kappa} C'}{P \vdash \langle (\mathcal{S}, \gamma) : C, s\rangle \xrightarrow{\kappa} (\mathcal{S}, \gamma) : C'}$$

$$\frac{\gamma \vdash \langle e\rangle \overset{\kappa}{\Rightarrow} \langle v_{d\ t}\rangle}{P \vdash \langle \gamma, \mathcal{S}[e]\rangle \xrightarrow{\kappa} \langle \gamma, \mathcal{S}[v_{d\ t}]\rangle}$$

Figure 4.7: Utility rules for statement evaluation

Statement evaluation rules

With the given tools we can define the set of tuples $P \vdash C^\circ \xrightarrow{\kappa} C^\bullet$. The program evaluation rule handles the case where C° denotes the program. The rule simply transitions into evaluating the body of the program and performs no actions:

$$\frac{}{P \vdash \langle P\rangle \rightarrow \langle \varepsilon, \mathsf{body}(P)\rangle}\ .$$

Statement evaluation rules are the most complicated ones and do the most work. Therefore, we present them in three parts. First, we look at utility rules that simplify the presentation of other rules. Then we look at the rules that evaluate statements, and finally the rules that handle function and system calls.

The utility rules are presented in Figure 4.7. The first two state that if it is possible to transition from $\langle C, s\rangle$ using action κ, then it is also possible to transition from configuration C that is extended with another stack frame $(\mathcal{S}, \gamma)$ using the same action. Intuitively, these rules allow us to specify all other rules without having to manipulate the entire configuration, and we can focus only on the parts of the configuration that are actually relevant. The third utility rule states that if an expression e can be evaluated to value $v_{d\ t}$ using action κ then statement $\mathcal{S}[e]$ can be evaluated to statement $\mathcal{S}[v_{d\ t}]$ using the same action.

Regular statement evaluation rules are presented in Figure 4.8. Mostly, the rules are straightforward and similar to how they are usually presented for WHILE-language small-step operation semantics. For instance, the rule for the while-statement transitions into evaluating an if-statement that contains the original statement, and for the skip-statement we transition into the environment as a result.

The rule that handles return-statements matches the statement $\mathcal{S}[\texttt{return}\ v_{d\ t}]$. This corresponds to the case where we reach a return statement inside a function body during evaluation. This rule correctly handles the evaluation of statements such as $(\texttt{return}\ e;\ s_2);\ s_3$ by dropping s_2 and s_3. Other than that, the rule takes the previous stack frame $(\mathcal{S}', \gamma')$, plugs the returned value $v_{d\ t}$ into the evaluation context $\mathcal{S}'$ and continues evaluating the restored statement $\mathcal{S}'[v_{d\ t}]$ using the environment γ'.

$$\frac{}{P \vdash \langle \gamma, \mathsf{skip} \rangle \to \gamma}$$

$$\frac{P \vdash \langle \gamma, s_1 \rangle \xrightarrow{\kappa} \gamma'}{P \vdash \langle \gamma, s_1;\ s_2 \rangle \xrightarrow{\kappa} \langle \gamma', s_2 \rangle} \quad \frac{P \vdash \langle \gamma, s_1 \rangle \xrightarrow{\kappa} \langle \gamma', s_1' \rangle}{P \vdash \langle \gamma, s_1;\ s_2 \rangle \xrightarrow{\kappa} \langle \gamma', s_1';\ s_2 \rangle}$$

$$\frac{\gamma' = \gamma[x \mapsto \bot_{d\,t}]}{P \vdash \langle \gamma, \{x : d\,t;\ s\} \rangle \to \langle \gamma', s \rangle} \quad \frac{\gamma' = \gamma[x \mapsto v_{d\,t}]}{P \vdash \langle \gamma, x = v_{d\,t} \rangle \to \gamma'}$$

$$\frac{}{P \vdash \langle (\mathcal{S}', \gamma') : \gamma, (\mathsf{return}\, v_{d\,t};\ s_1);\ \ldots;\ s_n \rangle \to \langle \gamma', \mathcal{S}'[v_{d\,t}] \rangle}$$

$$\frac{}{P \vdash \langle \gamma, \mathsf{if}\ \mathsf{true}^{+}_{\mathsf{bool}}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rangle \to \langle \gamma, s_1 \rangle}$$

$$\frac{}{P \vdash \langle \gamma, \mathsf{if}\ \mathsf{false}^{+}_{\mathsf{bool}}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rangle \to \langle \gamma, s_2 \rangle}$$

$$\frac{}{P \vdash \langle \gamma, \mathsf{while}\ e\ \mathsf{do}\ s \rangle \to \langle \gamma, \mathsf{if}\ e\ \mathsf{then}\ (s;\ \mathsf{while}\ e\ \mathsf{do}\ s)\ \mathsf{else}\ \mathsf{skip} \rangle}$$

Figure 4.8: Statement evaluation rules

Function and system call evaluation rules

Function and system call evaluation rules are presented in Figure 4.9. They are used when a function call with fully evaluated arguments is found in statement evaluation context. In that case, we look up the function instance (ℓ, θ) using argument domains and types, and continue evaluating the body of the function $\mathsf{body}^{\theta}_{P}(\ell)$ with environment γ' mapping parameters x_i to the arguments $v^i_{d_i\ t_i}$. The statement evaluation context $\mathcal{S}$ and the state of the current environment γ are saved before continuing with evaluation.

System calls are handled similarly to function calls. We also perform an instance lookup but this time we construct action κ of the form

$$v = f_{(k,t),(k_1,t_1),\ldots,(k_n,t_n)}(d, d_1, \ldots, d_n, v^1, \ldots, v^n) \ ,$$

where v^i is the monomorphic value of the i-th function argument, v is the value returned by the call, d is the protection domain of the result and d_i is the protection domain of the i-th argument. Unlike function calls, system calls do not need to store the stack frame and can simply continue with evaluating the statement where the call expression has been replaced by the value returned by the call. Note that we could have handled system calls with expression evaluation rules but chose to present them here instead due to the similarity to function calls.

$$\frac{\begin{array}{c}(\ell, \theta) = \mathsf{instance}_P(f; d_1\, t_1, \ldots, d_n\, t_n) \quad s = \mathsf{body}^\theta_P(\ell) \\ \gamma' = \varepsilon[x_i \mapsto v^i_{d_i\ t_i} \mid (x_i : d_i\, t_i) \in \mathsf{arg}^\theta_P(\ell)]\end{array}}{P \vdash \langle \gamma, \mathcal{S}[f(v^1_{d_1\ t_1}, \ldots, v^n_{d_n\ t_n})] \rangle \to \langle (\mathcal{S}, \gamma) : \gamma', s \rangle}$$

$$\frac{\begin{array}{c}(\ell, \theta) = \mathsf{instance}_P(f; d_1\, t_1, \ldots, d_n\, t_n) \quad \bot = \mathsf{body}^\theta_P(\ell) \\ d\, t = \mathsf{ret}^\theta_P(\ell) \quad k = \mathsf{kind}(d) \quad k_i = \mathsf{kind}(d_i) \\ \kappa \equiv (v = f_{(k,t),(k_1,t_1),\ldots,(k_n,t_n)}(d, d_1, \ldots, d_n, v^1, \ldots, v^n))\end{array}}{P \vdash \langle \gamma, \mathcal{S}[f(v^1_{d_1\ t_1}, \ldots, v^n_{d_n\ t_n})] \rangle \xrightarrow{\kappa} \langle \gamma, \mathcal{S}[v_{d\ t}] \rangle}$$

Figure 4.9: Function and system call evaluation rules

System calls have names of the form $f_{(k,t),(k_1,t_1),\ldots,(k_n,t_n)}$, where k_i is the protection domain kind of the i-th argument and k is the PDK of the output. As we already saw with evaluation of addition, this is necessary because we pass the values from **Val** to system calls but the system calls need to perform differently based on the types of the values. We use protection domain kinds as indices, because domains of the same kind share the implementation. For example, every domain of the additive three-party scheme has the same algorithms for performing multiplication. It is also important to note that we pass system calls to the protection domains of the arguments. This is necessary because private operations generally need to perform in a different way depending on protection domains. For example, while all additive three-party domains have the same multiplication algorithm they need to send network messages to different parties depending on the specific protection domain.

4.4.3 Trace semantics

In this section we present the meaning of SecreC programs in terms of traces. A potentially infinite sequence of actions $\circ \xrightarrow{\kappa_1} \circ \xrightarrow{\kappa_2} \circ \ldots$, where $\kappa_i \in \mathcal{A}$, is called a *trace*. Traces may be empty, finite or infinite. For example, a configuration in the final state has an empty trace $\circ$, but trace $\circ \to \circ \to \ldots$ corresponds to programs that loop forever while performing no actions. Forgetting the intermediate states is useful when we want to reason about program transformations and optimizations. For example, we might want to show that program compilation does not change the set of possible traces the program may take. Formally, the set of traces $\mathcal{T} = \mathcal{A}^* \cup \mathcal{A}^\omega$ is defined as the union of finite and infinite strings over actions $\mathcal{A}$.

The *trace semantics* of program P is defined as a set of traces $[\![P]\!] \subseteq \mathcal{T}$ by collecting all possible finite and infinite execution traces starting with program configuration $\langle P \rangle$. Formally, we define the semantics of programs P as follows:

$$\begin{aligned} [\![P]\!] &= \{\circ \overset{\kappa_1}{\to} \circ \ldots \circ \overset{\kappa_n}{\to} \circ \mid P \vdash \langle P \rangle \overset{\kappa_1}{\to} C_1^\circ \wedge \ldots \wedge P \vdash C_n^\circ \overset{\kappa_n}{\to} C\} \\ &\cup \{\circ \overset{\kappa_1}{\to} \circ \ldots \mid P \vdash \langle P \rangle \overset{\kappa_1}{\to} C_1^\circ \wedge \ldots\} \ . \end{aligned}$$

Note that terminating programs must all eventually stop in some final configuration C because the body of the program may not contain return statements.

A *symbolic trace*, denoted with T, is a potentially infinite and potentially infinitely branching tree where each path, starting from the root vertex, corresponds to a trace that could be generated by the program. For program P we want a single symbolic trace to correspond to trace semantics $[\![P]\!]$ of the program. Let $\mathcal{N}_t$ be the set of names for each type t such that $\mathcal{N}_t \cap \mathcal{N}_{t'} = \emptyset$ iff $t \neq t'$. We can assume that the set of names is countably infinite for every type t. A *symbolic action* A is either an empty action or an action of the form

$$\mathsf{v} = f(d, d_1, \ldots, d_n, \mathsf{v}_1, \ldots, \mathsf{v}_n) \ ,$$

where $\mathsf{v} \in \mathcal{N}_t$, $\mathsf{v}_i \in \mathcal{N}_{t_i}$, and f is a system call with signature $\prod_{i=1}^{n}(k_i, t_i) \to (k, t)$ where $k_i = \mathsf{kind}(d_i)$ and $k = \mathsf{kind}(d)$.

Each node u of a symbolic trace T is labeled with a symbolic action A_u. The number of descendants of non-leaf nodes depends on the labeling action. For ein mpty action we have exactly one descendant. A node u labeled with a non-empty action A_u that performs a system call f with signature $\prod_{i=1}^{n}(k_i, t_i) \to (k, t)$ has only one descendant if k is non-public. However, if k is public then u has a descendant for each value of type t corresponding to the system call returning that specific value for given inputs. For each downward path in T, a symbolic value may occur in a non-public argument position of an action only after it has occurred in the result position of some action previously.

Let g be a mapping from names $\mathcal{N} = \bigcup \mathcal{N}_t$ to values $\mathbf{Val}$. For a symbolic trace T let $g(\mathsf{T}) \in \mathcal{T}$ denote the concrete trace where each symbolic value v is replaced by $g(\mathsf{v})$ and concrete branches are chosen corresponding to the returned values. Let $\lceil\!\lceil \mathsf{T} \rceil\!\rceil \subseteq \mathcal{T}$ be the set of traces $\{g(\mathsf{T})\}$ for all possible mappings g.

The semantics of program P defines a unique, up to α-conversion, symbolic trace $\lfloor\!\lfloor P \rfloor\!\rfloor$ which can either be defined similarly to the trace semantics or directly recovered from $[\![P]\!]$. It is easy to see that $\lceil\!\lceil \lfloor\!\lfloor P \rfloor\!\rfloor \rceil\!\rceil = [\![P]\!]$. It is also easy to see that up to α-conversion, $\lceil\!\lceil \cdot \rceil\!\rceil$ is an injective mapping and, thus, there exists essentially a single symbolic trace corresponding to the trace semantics $[\![P]\!]$.

4.5 Execution of programs

Our runtime environment implements an arithmetic black box (Section 2.3.4) with multiple protection domains. Its security is given through universal composability

(Section 2.3.1). In heavily simplified terms, a protocol suite is universally composable if its every protocol is secure on its own, and composing them with any other protocol, either sequentially or in parallel, does not weaken security.

In our case, for each type t and each protection domain, the arithmetic black box $\mathcal{F}_{\mathsf{ABB}}$ stores a partial mapping $\mathsf{st}_{t,d}$ from names $\mathcal{N}_t$ to values $\mathbf{Val}$. The ABB accepts instructions of the form $\mathsf{v} = f(d, d_1, \ldots, d_n, \mathsf{v}_1, \ldots, \mathsf{v}_n)$ where system call f has signature $\prod_{i=1}^{n}(k_i, t_i) \to (k, t)$, $\mathsf{v} \in \mathcal{N}_t$ and $\mathsf{v}_i \in \mathcal{N}_{t_i}$ such that $k = \mathsf{kind}(d)$ and $k_i = \mathsf{kind}(d_i)$. The ABB implements the system call. When receiving an operation in the aforementioned form, the ABB will look up the values of the handles v_i, and update $\mathsf{st}_{t,d}$ to map the resulting handle v to its proper value. In addition to values of the arguments, this may also depend on the responses $\mathcal{F}_{\mathsf{ABB}}$ receives from the environment and the random choices of $\mathcal{F}_{\mathsf{ABB}}$. Hence, the ABB refines the non-determinism present in the dynamic semantics of the language.

Given an n-party ABB, a program P is executed as follows. There are n computing parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, the ABB $\mathcal{F}_{\mathsf{ABB}}$, the environment $\mathcal{Z}$ and the adversary $\mathsf{Sim}^{\mathcal{A}}$. The outer environment $\mathcal{Z}$ may provide inputs and supply random values. Adversary $\mathsf{Sim}^{\mathcal{A}}$ may corrupt certain parties and perform other permitted attacks on $\mathcal{F}_{\mathsf{ABB}}$. All parties $\mathcal{P}_i$ give instructions to $\mathcal{F}_{\mathsf{ABB}}$ according to the symbolic trace $\llfloor P \rrfloor$. When an action needs to produce a public value, the descendant corresponding to that value is chosen. This way we select only a single trace from $\llbracket P \rrbracket$. In effect, this execution defines a probability distribution over the set of traces $\llbracket P \rrbracket$. We denote this distribution by $\mathcal{D}[\mathcal{F}_{\mathsf{ABB}}, \mathcal{Z}, \mathsf{Sim}^{\mathcal{A}}](P)$.

Let π_{ABB} denote the implementation of the ideal functionality $\mathcal{F}_{\mathsf{ABB}}$. There are a number of different implementations that are at least as secure as $\mathcal{F}_{\mathsf{ABB}}$, meaning that there exists a simulator Sim such that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$, the views of $\mathcal{Z}$ in $\mathcal{Z}\|\pi_{\mathsf{ABB}}\|\mathcal{A}$ and $\mathcal{Z}\|\mathcal{F}_{\mathsf{ABB}}\|\mathsf{Sim}^{\mathcal{A}}$ are indistinguishable. One possible implementation is the additive three-party sharing we will (partially) implement in Chapter 5. There are many alternatives like Shamir's scheme, GMW, Yao, FHE, or even SGX, depending on which classes of adversaries we want to be secure against.

We can now state the theorem of correctness and security for the execution of program P using the comparison of the real implementations of protocols and the specification. The proof of the theorem is trivial, showing that the universal composability framework is wholly adequate to relating the abstract and concrete execution of protocols, allowing us to concentrate on specifying of what should be executed.

Theorem 1. *Let π_{ABB} be a secure implementation of $\mathcal{F}_{\mathsf{ABB}}$. There exists a simulator* Sim *such that for any environment $\mathcal{Z}$ and adversary $\mathcal{A}$ from the class of adversaries admitted by π_{ABB}, it holds that:*

Correctness: *distributions of traces $\mathcal{D}[\mathcal{F}_{\mathsf{ABB}}, \mathcal{Z}, \mathsf{Sim}^{\mathcal{A}}](P)$ and $\mathcal{D}[\pi_{\mathsf{ABB}}, \mathcal{Z}, \mathcal{A}](P)$*

are indistinguishable; and

Security: *the views of the environment* $\mathcal{Z}$ *in systems* $\mathcal{Z}\|\mathcal{P}_1\|\ldots\|\mathcal{P}_n\|\mathcal{F}_{\mathsf{ABB}}\|\mathsf{Sim}^{\mathcal{A}}$ *and* $\mathcal{Z}\|\mathcal{P}_1\|\ldots\|\mathcal{P}_n\|\pi_{\mathsf{ABB}}\|\mathcal{A}$ *are indistinguishable.*

Proof. It follows directly from the fact that π_{ABB} is at least as secure as $\mathcal{F}_{\mathsf{ABB}}$. From the security of π_{ABB} we know that there exists a simulator Sim such that for every environment $\mathcal{Z}'$ and adversary $\mathcal{A}$ the views of $\mathcal{Z}'$ in systems $\mathcal{Z}'\|\mathcal{F}_{\mathsf{ABB}}\|\mathsf{Sim}^{\mathcal{A}}$ and $\mathcal{Z}'\|\pi_{\mathsf{ABB}}\|\mathcal{A}$ are indistinguishable. This also holds for the environment $\mathcal{Z}' = \mathcal{Z}\|\mathcal{P}_1\|\ldots\|\mathcal{P}_n$. Both claims of the theorem state the indistinguishability of certain parts of these views. □

4.6 Security of information flow

In this section we will show that non-declassifying system calls do not increase adversaries knowledge during program execution. This also means that if a program does not invoke any declassifying system calls then the adversary learns nothing new from executing this program.

Protection domains give us a simple discipline for information flow control. An observer able to access data only in certain protection domains will learn nothing about the data in other domains, as long as no operation explicitly transfers data between these protection domains.

The semantics of the language specifies the possible orders in which system calls can be made. The actual execution of the program depends on the implementations of these system calls. The concrete semantics of a system call depends on the arithmetic black box, the environment and the adversary. Given an n-ary system call f the semantics is given by a function:

$$[\![f]\!] : \mathbf{PD}^{n+1} \times \mathbf{Val}^n \times \mathcal{W} \to \mathbf{Val} \times \mathcal{W} \ ,$$

where $\mathbf{PD}$ is the set of all protection domains and $\mathcal{W}$ denotes the set of all possible states of the environment. The behavior of system calls does not only depend on their explicit inputs but also on the state of the outside environment influenced by $\mathcal{Z}$ and $\mathsf{Sim}^{\mathcal{A}}$. The execution of a system call influences the environment. We simulate this by returning, in addition to the resulting value, the state of the modified environment.

Let $\boldsymbol{\mathcal{W}}$ denote the initial distribution of the environment. This distribution is known to everyone. Note that the semantics of system calls are deterministic. The random coins that system calls use are considered to be part of $\mathcal{W}$. While the initial distribution $\boldsymbol{\mathcal{W}}$ is publicly known, the concrete initial environment, which would tell us the values of random coins, is not known.

To speak about information flow security, we partition the set of protection domains into low and high domains such that $\mathbf{PD} = \mathbf{PD}_L \uplus \mathbf{PD}_H$ and $\texttt{public} \in \mathbf{PD}_L$. Low domains contain the information that is publicly know and high domains information that is intended to be kept secret. Similarly the environment is split into low and high parts: $\mathcal{W} = \mathcal{W}_L \times \mathcal{W}_H$. These parts must be independent in the initial distribution $\boldsymbol{\mathcal{W}}$.

For each system call action $\kappa \equiv (v = f(d, d_1, \ldots, d_n, v_1, \ldots, v_n))$ we define its *low-slice* $\overline{\kappa}$ by replacing each input v_i by a placeholder $\star$ if $d_i \in \mathbf{PD}_H$, and replacing the output v by $\star$ if $d \in \mathbf{PD}_H$. This definition naturally extends to traces.

A non-empty action κ is called a *declassification action* if $d \in \mathbf{PD}_L$ but $d_i \in \mathbf{PD}_H$ for some $1 \leq i \leq n$. We require that the semantics of non-declassifying system calls behave as we expect in terms of information flow: the low parts of the result may only depend on the low parts of the input. Namely, the low part of the environment may not be influenced neither by high inputs nor by the high part of the input environment. It is natural to require the environment to ensure the absence of other information flows even without the cooperation of the adversary.

Definition 7. Let f be a system call with semantics

$$[\![f]\!] : \mathbf{PD}^{n+1} \times \mathbf{Val}^n \times \mathcal{W}_L \times \mathcal{W}_H \to \mathbf{Val} \times \mathcal{W}_L \times \mathcal{W}_H \ .$$

We say that f is *non-declassifying* whenever for every $j \in \{1, \ldots, n+1\}, d_j \in \mathbf{PD}$, $x_j \in \mathbf{Val}$, $x'_j \in \mathbf{Val}$, $(w^i_L, w'^i_L, w^o_L, w'^o_L) \in \mathcal{W}^4_L$, $(w^i_H, w'^i_H, w^o_H, w'^o_H) \in \mathcal{W}^4_H$, if

$$\begin{array}{ll} & [\![f]\!](d_1, ..., d_{n+1}, x_1, ..., x_n, w^i_L, w^i_H) = (x_{n+1}, w^o_L, w^o_H) \\ \wedge & [\![f]\!](d_1, ..., d_{n+1}, x'_1, ..., x'_n, w'^i_L, w'^i_H) = (x'_{n+1}, w'^o_L, w'^o_H) \\ \wedge & \forall k \in \{1, \ldots, n\} \text{ if } d_k \in \mathbf{PD}_L \text{ then } x_k = x'_k \\ \wedge & w^i_L = w'^i_L \end{array}$$

then $w^o_L = w'^o_L$ and if $d_{n+1} \in \mathbf{PD}_L$ then $x_{n+1} = x'_{n+1}$. Superscript i denotes input and o denotes output. Note that a system call with public inputs and output can also be non-declassifying if it does not reveal high inputs.

We use the term *low adversary* to denote an adversary that can only observe the low part of the environment and the low-slices of the execution trace. We will show that unless κ is a declassification action, the execution of κ does not increase the low adversary's knowledge about the high part of the initial state of the environment. We define a probabilistic notion of the knowledge the adversary has after observing the low-slice of the trace so far.

Definition 8. We say that an adversary's knowledge $\mathcal{KN}_P^{\overline{T}, w_L}$ is a probability distribution over pairs (w, T) denoting the probability that evaluation of program P started in the state $w \in \mathcal{W}$ proceeds along finite trace $T \sqsubseteq [\![P]\!]_w$, given that the low-slice of the trace so far has been observed to be $\overline{T}$ and the initial low part of the initial state was w_L.

Because we are talking about probabilities over trace prefixes, we assume that the set of actions $\mathcal{A}$ is measurable. This is a natural assumption as labels are essentially finite products of values and some countable sets. Of course, we also need to assume that the set of values $\mathbf{Val}$ and the environment $\mathcal{W}$ admit to measure.

The concept of low-slices naturally extends to expressions, statements, evaluation contexts, and configurations. They are straightforward to define by replacing all high values $v_{d\,t}$, where $d \in \mathbf{PD}_H$, with $\star_{d\,t}$. All low-slices are denoted using an overline. For example, the low-slice of a configuration C is denoted with $\overline{C}$. When writing $C_1^\circ \sim_L C_2^\circ$ we mean that the low-slices of the respective program configurations are equal: $\overline{C_1^\circ} = \overline{C_2^\circ}$. This notation naturally extends to actions, traces, and expressions. *Low equivalence* is naturally an equivalence relation.

Lemma 1. *Let P be a well-typed program. Let $P \vdash C_1^\circ \xrightarrow{\kappa_1} C_1^\bullet$ and $P \vdash C_2^\circ \xrightarrow{\kappa_2} C_2^\bullet$ be two possible transitions that may be taken during the program's execution. If $C_1^\circ \sim_L C_2^\circ$ and $\kappa_1 \sim_L \kappa_2$ then $C_1^\bullet \sim_L C_2^\bullet$.*

Proof. Notice that by taking the low-slice of a configuration we only replace values, meaning that low-slicing preserves structure. The choice of semantic rule depends almost always only on the structure of the program configuration. The only exception is the case of if-expressions, for which the rule that we apply depends on the value of the public conditional. Hence, the same transition rules apply if $C_1^\circ \sim_L C_2^\circ$.

We will prove the case for assignments in more detail. The rest of the proof is more succinct. Assume that C_1° is of the form $\langle \gamma_1, x = v_{d\,t} \rangle$. Because $C_1^\circ \sim_L C_2^\circ$ we know that C_2° has to be of the form $\langle \gamma_2, x = u_{d\,t} \rangle$ such that $\gamma_1 \sim_L \gamma_2$ and $u_{d\,t} \sim_L v_{d\,t}$. From the evaluation rule E-Assign that applies in the case of assignment, C_1° transitions into $C_1^\bullet = \gamma_1[x \mapsto v_{d\,t}]$ and C_2° into $C_2^\bullet = \gamma_2[x \mapsto u_{d\,t}]$. Neither transition performs any actions. Regardless of whether d is low or high: $C_1^\bullet = \gamma_1[x \mapsto v_{d\,t}] \sim_L \gamma_2[x \mapsto v_{d\,t}] \sim_L \gamma_2[x \mapsto u_{d\,t}] = C_2^\bullet$.

In the case of variable declarations the low equivalence of target environments can be shown exactly the same way. However, we also need to show that target statements are low equivalent. This is straightforward because by taking a low-slice, the structure of statements does not change. For sequencing-, while- and skip-statements the proof is trivial as the rules are purely structural. The case of if-statements is also simple as their behavior only depends on public values.

In the case of variable expression evaluation assume that C_1° is in of form $\langle \gamma_1, \mathcal{S}_1[x] \rangle$. We know that C_2° has to be in of form $\langle \gamma_2, \mathcal{S}_2[x] \rangle$ such that $\gamma_1 \sim_L \gamma_2$ and $\mathcal{S}_1 \sim_L \mathcal{S}_2$. If $d \in \mathbf{PD}_L$ then it follows that $u = v$ and the statement is trivial.

If d is a high domain then

$$C_1^\bullet = \langle \gamma_1, \mathcal{S}_1[u_{d\ t}] \rangle \sim_L \langle \gamma_1, \mathcal{S}_1[\star_{d\ t}] \rangle \sim_L \langle \gamma_2, \mathcal{S}_2[\star_{d\ t}] \rangle \sim_L \langle \gamma_2, \mathcal{S}_2[v_{d\ t}] \rangle = C_2^\bullet\ .$$

Constants and public addition expressions are handled similarly to variables. Let C_1° be of the form $\langle \gamma_1, \mathcal{S}_1[u^1_{d\ t} + v^1_{d\ t}] \rangle$ such that $d \in \mathbf{PD}_H$. We know that C_2° has to be of the form $\langle \gamma_2, \mathcal{S}_2[u^2_{d\ t} + v^2_{d\ t}] \rangle$ where $\gamma_1 \sim_L \gamma_2$ and $\mathcal{S}_1 \sim_L \mathcal{S}_2$. Let configuration C_i° perform action $\kappa_i \equiv (w^i = \mathsf{add}_{k,t}(d, u^i, v^i)$. In both cases $\overline{\kappa_i} \equiv (\star = \mathsf{add}_{k,t}(d, \star, \star))$. Finally, we see that

$$C_1^\bullet = \langle \gamma_1, \mathcal{S}_1[w^1_{d\ t}] \rangle \sim_L \langle \gamma_2, \mathcal{S}_2[\star_{d\ t}] \rangle \sim_L \langle \gamma_2, \mathcal{S}_2[w^2_{d\ t}] \rangle = C_2^\bullet\ .$$

The case of system calls is very similar to non-public addition but simply more involved as some of the arguments may also be public. The cases of function calls and function return are again straightforward and techniques that we have already seen can be applied. Note that if all system calls had to be deterministic then we could prove a more powerful result that does not require the low equivalence of labels as an assumption.

Finally, we also need to prove the induction step. Assume that the statement holds for a given $P \vdash C_1^\circ \xrightarrow{\kappa_1} C_1^\bullet$ and $P \vdash C_2^\circ \xrightarrow{\kappa_2} C_2^\bullet$ where $C_1^\circ \sim_L C_2^\circ$. There are a few possible cases of which rule to apply depending on the structure of the transitions. We will only look at one of them and the proof follows similarly for other cases.

Let C_1° be of the form $\langle C_1, s_1 \rangle$ and $C_1^\bullet$ of the form C_1'. From $C_1^\circ \sim_L C_2^\circ$ we know that C_2° is also of the form $\langle C_2, s_2 \rangle$, such that $C_1 \sim_L C_2$, and $s_1 \sim_L s_2$. From the induction assumption: $C_1' \sim_L C_2' = C_2^\bullet$ and transition label κ_1 and κ_2 are also low equivalent. To prove the induction step let $P \vdash \langle (\mathcal{S}_1, \gamma_1) : C_1, s_1 \rangle \xrightarrow{\kappa_1} (\mathcal{S}_1, \gamma_1) : C_1'$. Thus, $P \vdash \langle (\mathcal{S}_2, \gamma_2) : C_2, s_2 \rangle \xrightarrow{\kappa_2} (\mathcal{S}_2, \gamma_2) : C_2'$. From the induction assumptions and the low equivalence of the input configurations we immediately have that $(\mathcal{S}_1, \gamma_1) : C_1' \sim_L (\mathcal{S}_2, \gamma_2) : C_2'$. □

For each possible initial environment $w \in \mathcal{W}$ the semantics of system calls and program P uniquely determine an execution trace $T \in [\![P]\!]$. We denote this trace by $[\![P]\!]_w$. Let $\overline{\mathcal{A}} = \{\overline{\kappa} \mid \kappa \in \mathcal{A}\}$ be the set of all low-slices of transition labels and $\overline{\mathcal{T}} = \{\overline{T} \mid T \in \mathcal{T}\}$ the set of all low-slices of traces.

Definition 9. Let $\mathcal{I}_P^{\overline{T}, w_L}$ denote the set of all possible initial environments w' with the low part w_L such that the low-slice of $[\![P]\!]_{w'}$ has been thus far observed to be $\overline{T}$. Formally $\mathcal{I}_P^{\overline{T}, w_L} = \{w' \mid w'_L = w_L \wedge \overline{T} \sqsubseteq \overline{[\![P]\!]_{w'}}\}$ where $\sqsubseteq$ denotes that a sequence is a prefix of the other one.

Lemma 2. *Let $w_L \in \mathcal{W}_L$, $\overline{T} \in \overline{\mathcal{T}}$ and $\kappa \in \mathcal{A}$ such that κ is not a declassification action. Then either $\mathcal{I}_P^{\overline{T;\kappa}, w_L} = \mathcal{I}_P^{\overline{T}, w_L}$ or $\mathcal{I}_P^{\overline{T;\kappa}, w_L} = \emptyset$.*

Proof. We first note that the low-slice of the program context and the low part of the environment are the same for all $w' \in \mathcal{I}_P^{\overline{T},w_L}$ after the execution that produces the trace with low-slice $\overline{T}$. This is a straightforward implication of Lemma 1. All initial configurations $\langle P \rangle$ are trivially low equivalent and in every step low equivalent configurations transition into low equivalent ones. Similarly, the low part of the environment in different executions evolves in the same way due to the constraints placed on the semantics of system calls. All system calls f have deterministic semantics $[\![f]\!]$ and, most notably, system calls with high arguments may not influence the low part of the environment.

We will now show that the low-slice of the current program context C_1° and the low part of the current environment w determine the low-slice of the next transition label κ', unless the next label is a declassification label. First, whether the next transition is a system call or an empty action is determined by the structure of C° and the types of values. If C° performs a non-public addition then $\kappa' \sim_L (\star = \mathsf{add}_{k,t}(d, \star, \star))$ and kind k, type t and domain d are all determined by $\overline{C^\circ}$.

Let us now consider the case where C° performs a system call

$$\kappa' \equiv (v_0 = f(d_0, d_1, \ldots, d_n, v_1, \ldots, v_n)) \enspace .$$

Note that the system call's name f and protection domains d_i, for $0 \leq i \leq n$, are all determined by program P and C°. In the low-slice of κ' the arguments of f are also completely determined by the program and the low-slice of the configuration. We need to show that the result v_0 is also determined. If the system call f is public, as $d_i \in \mathbf{PD}_L$, then $[\![f]\!]$ has to be deterministic given the arguments $v_1, \ldots, v_n$ and the low-slice of the environment w_L. Hence, we can compute the result v_0 from those arguments. If $d_0 \in \mathbf{PD}_H$ then v_0 is replaced with $\star$ in the low-slice of κ' and is thus trivially determined. In all other cases κ' is a declassification label.

As we saw, the low-slice of the configuration $\overline{C^\circ}$ and the low part of the environment w_L are uniquely determined by program P and the observed low trace $\overline{T}$. The low-slice of the configuration in turn determines the low-slice of the next label $\overline{\kappa'}$. Thus we either have $\mathcal{I}_P^{\overline{T;\kappa},w_L} = \mathcal{I}_P^{\overline{T},w_L}$ if $\overline{\kappa} = \overline{\kappa'}$ or $\mathcal{I}_P^{\overline{T;\kappa},w_L} = \emptyset$ otherwise. □

Lemma 3. *Let $w_L \in \mathcal{W}_L$, $\overline{T} \in \overline{\mathcal{T}}$ and $\kappa \in \mathcal{A}$ such that κ is not a declassification label. Then $\mathcal{KN}_P^{\overline{T;\kappa},w_L} = \mathcal{KN}_P^{\overline{T},w_L}$.*

Proof. This follows immediately from the previous lemma. □

Theorem 2. *Let $w_L \in \mathcal{W}_L$, $\overline{T} \in \overline{\mathcal{T}}$ such that all labels in $\overline{T}$ are non-declassifying. Then $\mathcal{KN}_P^{\overline{T},w_L} = \mathcal{KN}_P^{\emptyset,w_L}$.*

Proof. By induction using the previous lemma. □

Theorem 2 shows that adversary's knowledge does not increase until the first declassifying system call. Lemma 3 is actually stronger. It claims that after declassification subsequent non-declassifying system calls also do not increase adversary's knowledge. We do not restrict how much information declassifying system calls release. They could either reveal everything adversary could learn or they might not reveal any new information.

Theorem 3. *Let P be a program with body of the form*

$$\{\texttt{x} : d\,\texttt{bool};\ \texttt{x} = \texttt{get\_bool}();\ s\}$$

such that statement s and functions defined by P do not contain declassifying system calls. Let π^b_{ABB}, parametrized with $b \in \{\texttt{true}, \texttt{false}\}$ be a secure implementation of $\mathcal{F}^b_{\mathsf{ABB}}$ that

- *gives non-declassifying semantics to non-declassifying system calls, and*
- *return b in response to* `get_bool()`.

Let $\mathcal{P}_1, \ldots, \mathcal{P}_n$ be the computing parties executing the program P (Section 4.5). Then for any adversay $\mathcal{A}$ admitted by π^b_{ABB} the views of any environment $\mathcal{Z}$ in configuration $\mathcal{Z}||\mathcal{P}_1||\ldots||\mathcal{P}_n||\pi^{\texttt{true}}_{\mathsf{ABB}}||\mathcal{A}$ and $\mathcal{Z}||\mathcal{P}_1||\ldots||\mathcal{P}_n||\pi^{\texttt{false}}_{\mathsf{ABB}}||\mathcal{A}$ are indistinguishable.

4.7 Monomorphic representation

The syntax of SecreC presented here is quite simplified compared to the real language but it still offers high-level features in the form of domain polymorphism. The semantics of the language reflects the meaning but in this case presents evaluation in quite unrealistic manner. In this section we describe a more simplified language that models the compiler's intermediate representation. The dynamic semantics of the simplified language reflects the actual evaluation better.

There are two issues with the high-level language's evaluation rules. First, we are keeping track of protection domains and types of all values and variables. Second, function calls are dynamically dispatched by performing an instance lookup for every call. In reality we do not need to keep track of types during running time and all dynamic dispatches can be statically resolved.

We denote the syntactic elements of the monomorphic language using the same notation that we used for the high-level language. The languages are syntactically almost identical and it is always clear from the context whether we are talking about the polymorphic or the monomorphic one. The only difference is that

$$\frac{P; \emptyset \vdash \mathsf{body}(P) \quad \forall F_i \in P.\, P \vdash F_i}{\vdash P} \ (\textsc{MonoProgram})$$

Figure 4.10: Monomorphic program type checking rule

$$\frac{}{P \vdash f(x_1 : d_1\, t_1, \ldots, x_n : d_n\, t_n) : d\, t} \ (\textsc{MonoSysCall})$$

$$\frac{P; (\texttt{return} : d\, t, x_1 : d_1\, t_1, \ldots, x_n : d_n\, t_n) \vdash s}{P \vdash f(x_1 : d_1\, t_1, \ldots, x_n : d_n\, t_n) : d\, t\; s} \ (\textsc{MonoFunction})$$

Figure 4.11: Monomorphic function declaration type checking rules

the monomorphic language may not contain polymorphic functions and additive expressions are annotated with types (Equation (4.3)). The syntax for programs, data types and statements remains unmodified.

$$\begin{array}{rcl} F & ::= & f(x_1 : d_1\, t_1, \ldots, x_n : d_n\, t_n) : d\, t\; [s] \\ e & ::= & x \mid c_t \mid e_1 +_{d\, t} e_2 \mid f(e_1, e_2, \ldots, e_n) \end{array} \tag{4.3}$$

4.7.1 Static semantics

The type checking rules for the monomorphic language differ from the rules for the polymorphic language in a few aspects. First, the monomorphic language type checking rules are no longer defined via the existence of an instantiation context. Second, during function calls we no longer have to look up instances and instead look for a function with a concrete type. Lastly, functions may no longer be overloaded by type. Apart from that, the rules are fairly similar. In fact, the rules for statements are identical in both languages (see Figure 4.5), and hence, will not be repeated here. A monomorphic program P is well-typed if its body and every function definition is well-typed. This is reflected by the type checking rule in Figure 4.10. Procedures are well-typed if their body is well-typed given appropriately assigned types in the environment Γ (Figure 4.11). System calls are always well-typed.

Typing rules for expressions are presented in Figure 4.12. The only rule that is different from the polymorphic language is for function calls (the MonoCall rule). In the monomorphic case we simply check if a function with the given name exists. The lookup provides us with the argument and return types of the function. Then we check that the arguments have matching types. We write $P \vdash f : (d_1\, t_1, \ldots, d_n\, t_n) \to d\, t$ if program P declares an n-ary function or a system call named f and it has the corresponding signature.

$$\frac{(x : d\,t) \in \Gamma}{P; \Gamma \vdash x : d\,t}\ (\textsc{MonoVar}) \qquad \frac{}{P; \Gamma \vdash c_t : t^+}\ (\textsc{MonoLit})$$

$$\frac{P; \Gamma \vdash e_1 : d\,t \quad P; \Gamma \vdash e_2 : d\,t \quad t \in \{\texttt{int}, \texttt{int[]}\}}{P; \Gamma \vdash e_1 +_{d\,t} e_2 : d\,t}\ (\textsc{MonoAdd})$$

$$\frac{\begin{array}{c} P \vdash f : (d_1\,t_1, \ldots, d_n\,t_n) \to d\,t \\ P; \Gamma \vdash e_1 : d_1\,t_1 \quad \ldots \quad P; \Gamma \vdash e_n : d_n\,t_n \end{array}}{P; \Gamma \vdash f(e_1, \ldots, e_n) : d\,t}\ (\textsc{MonoCall})$$

Figure 4.12: Monomorphic language expression type checking rules

4.7.2 Dynamic semantics

Just like with static semantics, the dynamic semantics of the monomorphic language is largely similar to the polymorphic case presented in Section 4.4. Recall that in the polymorphic setting, values were annotated with protection domains and types. In the monomorphic case this is no longer so. For example, the environment γ now maps names directly to $\mathbf{Val}$.

Firstly, we need to extend expressions with values: $e ::= \ldots \mid v$ where $v \in \mathbf{Val}$. Expression and statement evaluation context can be defined similarly to the polymorphic case with the difference that values are no longer annotated with type information. Thus, we will not repeat the definitions here and refer the reader to Equations (4.1) and (4.2). Plain configurations C, program configurations C°, and target configurations $C^\bullet$ are all defined in a similar way.

Dynamic semantics are presented in small-step operational style using the set of tuples $P \vdash C^\circ \xrightarrow{\kappa} C^\bullet$. The presentation is again almost identical to the polymorphic case. We will highlight the rules for evaluating addition, function calls and system calls (Figure 4.13). For an addition expression, the type and protection domain are encoded in the operation itself and not in the value. The notable difference for function and system calls is that in the monomorphic setting we no longer need to perform dynamic instance lookup.

The trace semantics of monomorphic language program $[\![P]\!]$ is defined in the same manner as for the polymorphic language. We are reusing the same notion of traces as previously. This is possible because we have defined traces so that they hide the details of program state and only consider the actions that the program performs during its execution.

$$\frac{d = \mathtt{public} \quad w = u + v}{\gamma \vdash \langle u +_{d\,t} v \rangle \Rightarrow \langle w \rangle} \qquad \frac{d \neq \mathtt{public} \quad \kappa \equiv (w = \mathsf{add}_{\mathsf{kind}(d),t}(d, u, v))}{\gamma \vdash \langle u +_{d\,t} v \rangle \overset{\kappa}{\Rightarrow} \langle w \rangle}$$

$$\frac{s = \mathsf{body}_P(f) \quad \gamma' = \varepsilon[x_i \mapsto v_i \mid (x_i : d_i\, t_i) \in \mathsf{arg}_P(f)]}{P \vdash \langle \gamma, \mathcal{S}[f(v_1, \dots, v_n)] \rangle \to \langle (\mathcal{S}, \gamma) : \gamma', s \rangle}$$

$$\frac{\begin{array}{c} P \vdash f : (d_1\, t_1, \dots, d_n\, t_n) \to d\, t \\ \bot = \mathsf{body}_P(f) \quad k = \mathsf{kind}(d) \quad k_i = \mathsf{kind}(d_i) \\ \kappa \equiv (v = f_{(k,t),(k_1,t_1),\dots,(k_n,t_n)}(d, d_1, \dots, d_n, v_1, \dots, v_n)) \end{array}}{P \vdash \langle \gamma, \mathcal{S}[f(v_1, \dots, v_n)] \rangle \overset{\kappa}{\to} \langle \gamma, \mathcal{S}[v] \rangle}$$

Figure 4.13: Monomorphic language expression evaluation rules

$$\frac{\begin{array}{c} \forall (f, \ell, \theta) \in I : P; \theta; I \vdash f^{\ell} \rightsquigarrow F_{(f,\ell,\theta)} \\ F_I = \{F_i \mid i \in I\} \quad P; \emptyset; I \vdash \mathsf{body}(P) \rightsquigarrow s' \end{array}}{I \vdash P \rightsquigarrow \mathsf{pdk}(P)\ \mathsf{pd}(P)\ F_I\ s'} \ (\textsc{TransProgram})$$

Figure 4.14: Program translation rule

4.7.3 Monomorphization

We have defined the syntax and semantics of the monomorphic language. Here we cover translation from the polymorphic language into the monomorphic one. The type-directed translation rules model how the compiler generates lower level intermediate code from the high-level program.

For an instantiation context I, we write $I \vdash P \rightsquigarrow P'$ to mean that polymorphic program P translates into the monomorphic program P'. The top-level translation rule is presented in Figure 4.14. Every function instance in I and the body of program P is translated into monomorphic form and the resulting program is constructed out of the translated instances. The protection domain kind and protection domain declarations are taken directly from P.

A single polymorphic function may be translated into multiple functions, each with a concrete type. To distinguish between those functions we add some extra information to function names during translation. Namely, function f^{ℓ} with a concrete substitution θ will get translated into a function with name $f_{(\ell,\theta)}$. Function argument types, result types and the function body are substituted with θ to use concrete domains. Finally, the function body is translated with a properly extended environment. The function and system call translation rules are presented in Figure 4.15.

$$\frac{\mathsf{body}^{\theta}_{P}(\ell) = \bot}{P; \theta; I \vdash f^{\ell} \rightsquigarrow f_{(\ell,\theta)}(\mathsf{arg}^{\theta}_{P}(\ell)) : \mathsf{ret}^{\theta}_{P}(\ell)} \text{ (TransSysCall)}$$

$$\frac{\begin{array}{c}\mathsf{body}^{\theta}_{P}(\ell) \neq \bot \\ P; (\mathtt{return} : \mathsf{ret}^{\theta}_{P}(\ell), \mathsf{arg}^{\theta}_{P}(\ell)); I \vdash \mathsf{body}^{\theta}_{P}(\ell) \rightsquigarrow s'\end{array}}{P; \theta; I \vdash f^{\ell} \rightsquigarrow f_{(\ell,\theta)}(\mathsf{arg}^{\theta}_{P}(\ell)) : \mathsf{ret}^{\theta}_{P}(\ell)\ s'} \text{ (TransFunction)}$$

Figure 4.15: Function declaration translation rules

$$\frac{}{P; \Gamma; I \vdash v_{d\,t} : d\,t \rightsquigarrow v} \text{ (TransVal)}$$

$$\frac{(x : d\,t) \in \Gamma}{P; \Gamma; I \vdash x : d\,t \rightsquigarrow x} \text{ (TransVar)} \qquad \frac{}{P; \Gamma; I \vdash c_t : t^{+} \rightsquigarrow c_t} \text{ (TransLit)}$$

$$\frac{\begin{array}{c}t \in \{\mathtt{int}, \mathtt{int[]}\} \\ P; \Gamma; I \vdash e_1 : d\,t \rightsquigarrow e_1' \quad P; \Gamma; I \vdash e_2 : d\,t \rightsquigarrow e_2'\end{array}}{P; \Gamma; I \vdash e_1 + e_2 : d\,t \rightsquigarrow e_1' +_{d\,t} e_2'} \text{ (TransAdd)}$$

$$\frac{\begin{array}{c}P; \Gamma; I \vdash e_1 : d_1\,t_1 \rightsquigarrow e_1' \quad \ldots \quad P; \Gamma; I \vdash e_n : d_n\,t_n \rightsquigarrow e_n' \\ (\ell, \theta) = \mathsf{instance}_P(f; d_1\,t_1, \ldots, d_n\,t_n) \quad (f, \ell, \theta) \in I\end{array}}{P; \Gamma; I \vdash f(e_1, \ldots, e_n) : d\,t \rightsquigarrow f_{(\ell,\theta)}(e_1', \ldots, e_n')} \text{ (TransCall)}$$

Figure 4.16: Expression translation rules

Expression translation rules are given of the form $P; \Gamma; I \vdash e : d\,t \rightsquigarrow e'$ in Figure 4.16. The rules state that a high-level language expression e has type $d\,t$ and translates to monomorphic expression e'. The rules are very similar to the high-level language type checking rules. We only note that function calls are translated to refer to concrete instances of polymorphic functions. We will omit statement translation rules as they are completely structural and correspond directly to the rules from Figure 4.5 directly.

The translation rules are in one-to-one correspondence with the type checking rules. In fact, whenever a program is well-typed, there exists a monomorphic program into which it translates. The reverse also holds. If we can translate a program to monomorphic form it needs to be well-typed.

Proposition 2. *For every polymorphic program P and instantiation context I we* *have $I \vdash P$ iff there exists P' such that $I \vdash P \leadsto P'$.*

Proof. This follows immediately from the one-to-one correspondence between type checking and translation rules. If $I \vdash P$ then we can follow the translation rules without getting stuck, and hence, there must exists a P' such that $I \vdash P \leadsto P'$. If there exists a P' such that $I \vdash P \leadsto P'$ then we can take the rules the translation follows and simply erase from them the translation part and keep the type checking part. □

We can show that the translation only produces well-typed monomorphic programs.

Proposition 3. *For every program P and instantiation context I if $I \vdash P \leadsto P'$ then $\vdash P'$.*

Proof. If $I \vdash P \leadsto P'$ then we know that P has to be well-typed $I \vdash P$. When attempting to type check the body of P', we follow the exact same path as we do when type checking the body of P. However, when visiting function calls we will instead call concrete instances. From the rule TransCall we know that a well-typed instance must exist. Hence, the respective calls in the body of P' have to also exist because of the way translation of programs (TransProgram) is defined. □

4.7.4 Correctness of translation

To show the correctness of translation to monomorphic form we will show that equivalent polymorphic and monomorphic program configurations take the same transition steps. For this, we need to define what we mean by equivalent program configurations.

Environments γ and γ' are equivalent when erasing types $d\ t$ in γ gives γ'. We write this $\gamma \equiv \gamma'$. Let $\Gamma(\gamma)$ denote a type checking environment such that for every x, where $\gamma(x) = v_{d\ t}$ for some $v \in \mathbf{Val}$, we have $(x : d\ t) \in \Gamma(\gamma)$. Essentially, we map the dynamic language runtime variable environment γ to a type environment. Two configurations are equivalent if their components are equivalent and a high-level language statement translates to a low-level language statement in the respective context. Formally:

$$\frac{\gamma \equiv \gamma' \quad P; \Gamma(\gamma); I \vdash s \leadsto s' \quad P; I \vdash C \equiv C'}{P; I \vdash \langle C : \gamma, s\rangle \equiv \langle C' : \gamma', s'\rangle} \quad .$$

Lemma 4 (Bisimilarity)**.** *Let P be a program and I an instantiation context such that $I \vdash P \leadsto P'$. For every two equivalent program configurations $P; I \vdash C^\circ \equiv C'^\circ$ the following conditions hold:*

1. *For every label κ and polymorphic language target configuration $C^{\bullet}$ if $P \vdash C^{\circ} \overset{\kappa}{\rightarrow} C^{\bullet}$ then there exists $C'^{\bullet}$ such that $P' \vdash C'^{\circ} \overset{\kappa}{\rightarrow} C'^{\bullet}$ and $P; I \vdash C^{\bullet} \equiv C'^{\bullet}$.*

2. *For every label κ and monomorphic language target configuration $C'^{\bullet}$ if $P' \vdash C'^{\circ} \overset{\kappa}{\rightarrow} C'^{\bullet}$ then there exits $C^{\bullet}$ such that $P \vdash C^{\circ} \overset{\kappa}{\rightarrow} C^{\bullet}$ and $P; I \vdash C^{\bullet} \equiv C'^{\bullet}$.*

Proof. For a given program P and instance context I, the configuration C° determines C'°. Given κ and C°, the configuration $C^{\bullet}$ is also uniquely fixed. Similarly, given κ and C'°, the target configuration $C'^{\bullet}$ is uniquely fixed. These facts reduce the proof to induction over high-level and low-level statement evaluation rules for the first and second part respectively.

□

Theorem 4. *For every program P and instantiation context I if there exists P' such that $I \vdash P \rightsquigarrow P'$ then $[\![P]\!] = [\![P']\!]$. In other words the semantics are preserved under translation.*

Proof. This follows from bisimilarity. For program P let $I \vdash P \rightsquigarrow P'$. Consider a single trace $T \in [\![P]\!]$. The trace T follows from the initial configuration $\langle P \rangle$ taking steps $\kappa_1, \kappa_2, \ldots$. The program P and trace T together fix the intermediate configurations uniquely, and thus, by application of the bisimilarity lemma we get that $T \in [\![P']\!]$. By similar reasoning and by applying the second part of the bisimilarity lemma we also get that if $T \in [\![P']\!]$ then $T \in [\![P]\!]$. □

CHAPTER 5

A LANGUAGE FOR LOW-LEVEL SECURE MULTI-PARTY COMPUTATION PROTOCOLS

SHAREMIND features a large set of secure primitives: basic arithmetic, comparisons, bit extraction, bit conversions, division of arbitrary-width integers, as well as a full set of floating-point [62] and fixed-point [30] operations, including the implementations of elementary functions.

More often than not SHAREMIND protocols are specified in a compositional style forming a hierarchy, with more complex protocols invoking simpler ones. This is a very natural approach to software development. For example, floating-point operations use fixed-point operations which in turn use integer operations [71]. The choice to expand the set of primitives in SHAREMIND has been validated by the multitude of privacy-preserving applications it has been used for (see Chapter 6 for examples).

The implementation and maintenance of such a large set of protocols has turned out to be an error-prone and repetitive task. Manual attempts to optimize complex protocols over composition boundaries is a laborious task, prone to introduce errors and make the library of protocols unmaintainable. Implementation is made more difficult due to the fact that protocols need to work for several integer widths, and many abstractions in the implementation language, such as virtual function calls in C++/Java, entail an unacceptable runtime overhead. The task of building and maintaining implementations of protocols is naturally answered by introducing a domain-specific language (DSL) for specifying them.

In this chapter we describe the protocol DSL. It was designed by the author of this work, who was also the principal developer of the compiler. We discuss the design rationale of the language, show how it enables us to easily build protocols for complex operations, and describe our experience in using it, both in terms of performance and maintainability. We start with an informal overview of the language through a number of examples, allowing us to give the reader an intuitive understanding of the language and to highlight the important details. The design of the protocol language has been heavily influenced by Cryptol [83]—a language for specifying cryptographic algorithms. Syntactically our protocol language is quite different but both languages encode sizes in type level, are intended for cryptographic algorithm specification and both compile to a low-level circuit description. However, our protocol language is much more specialize and contains network communication primitives.

5.1 Language overview

We sought to create a programming language that facilitates the implementation of SMC protocols in a style similar to their specification in [22] and [62]. Our experience in implementing SMC protocols in C++ showed that the aspect most hindering our productivity and performance was the lack of composability.

Namely, our C++ protocol development framework is designed to allow a single protocol to be executed on many inputs at a time (SIMD style) but vectorized execution of two different protocols is only realizable by interleaving both protocols and manually packing all network messages together. Doing so is very time-consuming even for medium-sized protocols. Lack of composability leads to unreadable protocols, and difficulties with maintenance and introducing modifications. Fixing a bug means making changes in every place the modified protocol has been copied to. This has often led to sacrificing performance for readability and development time to an unacceptable degree.

Hence, the key design principle of the protocol DSL is to always put composability first: whenever the data-flow dependencies allow, the protocols are executed in parallel[1], no matter the order they occur in the code. For performance reasons it is important to keep the round complexity of protocols low, as a network roundtrip is orders of magnitude slower than the time it takes to evaluate the (arithmetic) gates in the protocols.

When implementing protocols in C++, large parts of the code dealt with how network messages are packed, and how and when they are sent to other computing parties. We decided to automate this process. In the protocol language we have simplified the network messages aspect and the programmer only has to specify what values are used by other parties, but not how or when they get there. The compiler minimizes the round count, chooses how values are packed into messages,

[1]We do not mean parallelism in the sense of using multiple cores but in the sense of packing many different values into one message and thus sending fewer network messages and reducing the number of communication rounds.

Listing 5.1: Typical top-level structure of a DSL protocol (add.prot)

```
parties 3

// Polymorphic function that adds two n-bit integers
def add(u: uint[n], v: uint[n]): uint[n] = u + v

/* Generate the following protocols: */
protocol add32(x: uint[32], y: uint[32]): uint[32] = add(x, y)
protocol add64(x: uint[64], y: uint[64]): uint[64] = x + y
```

and deals with sending and receiving network messages automatically.

We also wanted to have an optimizing compiler for the language so that the programmer does not have to manually optimize when compositions introduce possibilities for it. The protocol DSL is a functional language that enables writing code in a declarative style and lets the programmer manipulate protocols in a higher-order manner. For instance, it is natural to apply a protocol to each element of a vector by using a higher-order mapping operator.

The language supports type level integers, called *size types*, and arithmetic on them. Functions polymorphic in the number of input bits allow a protocol to be specified once for all input lengths. The language is not strongly coupled with Sharemind and can be used by other SMC frameworks. However, protocol DSL is mainly targeted for implementing arithmetic secret sharing schemes and only works with statically fixed number of parties.

5.1.1 Top-level structure

Typically, a program in the DSL declares the number of parties the protocols operate on, imports some external modules to use, defines some functions, and finally defines some protocols. Code will be generated only for protocol declarations and not for function definitions. Functions in the DSL may be polymorphic, for example, they may operate on arbitrary bit width integers, but protocols are strictly monomorphic and first-order.

A simple example to demonstrate the top-level structure of the language is presented in Listing 5.1. In the example we operate with 3 computing parties. The default mode of operation in the protocol DSL is that every computing party executes the same code. In this example all three parties perform exactly the same way. After declaring the number of parties, we define a function named add that takes two n-bit unsigned integers u and v, and returns an n-bit integer. The function is defined to evaluate to the sum of the arguments. The type of n-bit integers is denoted with **uint**[n] where the type argument n is placed between square brackets.

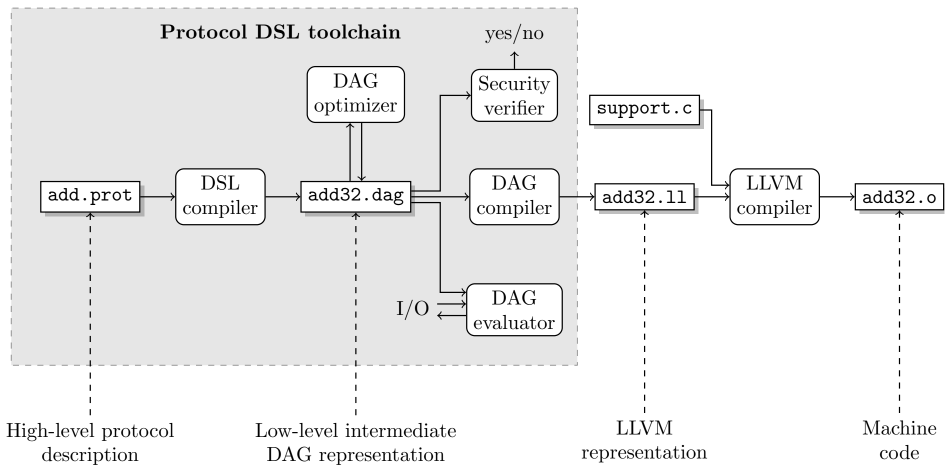


Figure 5.1: Overview of the protocol DSL toolchain

As the last step we declare that code is to be generated for two protocols: 32-bit and 64-bit addition. Notice that protocol declarations operate on concrete bit widths.

Figure 5.1 gives an overview of how the protocol DSL toolchain can be used to compile, evaluate or analyze the current example. Compiling the file will produce an intermediate code file for each protocol declarations (see Section 5.4). For the current example, two files `add32.dag` and `add64.dag` are produced. Each of the generated DAG files can be optimized, evaluated for testing, analyzed for security, and can be compiled to machine code via the LLVM toolchain. The object code can be linked in Sharemind to provide an automatically vectorized addition protocol. Object code can also be directly evaluated, used for testing or could be used by other SMC systems.

5.1.2 Implementing protocols for additive secret sharing

As an example we will define a basic protocol set for three-party additive secret sharing. Recall that values are shared over a finite ring. Different rings of the form $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$ can be used. The protocols perform operations on secret shared values, receiving the shares of the operands as inputs and delivering the shares of the results. A secret shared value $x \in R$, where R is one of the rings $\mathbb{Z}_{2^n}$ or $\mathbb{Z}_2^n$, is represented as a triple (x_1, x_2, x_3) with party $\mathcal{P}_i$ holding $x_i \in R$, satisfying $x_1 + x_2 + x_3 = x$, where addition is in ring R. A useful property of the additive scheme is that integers can be added with no network communication by adding the respective shares. Most other operations require at least one communication round. This means that the previous example in Listing 5.1 actually implements addition protocol for $\mathbb{Z}_{2^n}$. Next, we take a look at examples using network communication.

Listing 5.2: Additive resharing

```
def reshare(u: uint[n]): uint[n] = {
  let
    r = rng()
    w = u + r - (r from Next);
  w
}
```

Resharing

To ensure that shares are independent of values that an adversary may have previously observed, *resharing* is commonly used when implementing three-party additive protocols. Resharing operation takes input shares and returns new shares that are independent of input shares but reconstruct to the same value that input shares reconstruct to. The new shares can be used used as inputs for further protocols.

The implementation of resharing [23, Alg. 1] in our DSL is depicted in Listing 5.2. The i-th party receives the share u_i of some private value $u \in \mathbb{Z}_{2^n}$ as input. In order to obtain the output share u'_i, the i-th party generates a random value r_i, sends it to the previous party and finally adds to the input the difference between the generated random value and the random value that it has received from the next party. All arithmetic is performed modulo 2^n. The final shares obtained in the protocol will be $(u'_1, u'_2, u'_3) = (u_1 + r_1 - r_2, u_2 + r_2 - r_3, u_3 + r_3 - r_1)$. We see that $u'_1 + u'_2 + u'_3 = u_1 + u_2 + u_3 = u$, i.e. the output value is the same as the input value but u'_i are independent of u_i.

We define variable `reshare` of type `uint[n] -> uint[n]` denoting a function that takes an n-bit unsigned integer to an n-bit unsigned integer. The function argument is called `u` and its type `uint[n]` is indicated after a colon. The body of the function follows after the equality sign and it first randomly generates a value `r`, and computes `u + r - (r from Next)` as its result. For i-th party the expression `r` evaluates to the i-th share of `r` but the expression `r from Next` evaluates to $r_{\mathsf{n}(i)}$ where $\mathsf{n}(i)$ denotes the next index ($\mathsf{p}(\cdot)$ maps 1 to 2, 2 to 3 and 3 to 1).

It is important to note that we have not explicitly stated which computing party performs which computation. Recall that implicitly every computing party executes the same code. For instance, each party generates a random value independently even though it is only written once in the code. The types of variables are derived via type inference: from the type of the function we know that the input variable `u` must be an n-bit integer, and because addition operates on integers of equal bit width we know that the randomly generated variable `r` must also be an n-bit integer.

Sometimes it is useful to reshare a value in such a way that one of the parties holds 0 as its share. Such resharing protocol is given in Listing 5.3. In this protocol, Listing 5.3: Resharing between two parties

```
1   def reshareToTwo(u: uint[n]): uint[n] = {
2     let {1}
3       r2 = rng()
4       r3 = u - r2;
5
6     party:
7       1 -> 0
8       2 -> u + (r2 from 1)
9       3 -> u + (r3 from 1)
10  }
```

given the shares of an input $u = (u_1, u_2, u_3)$, the first computing party generates a random value r_2, sends it to the second computing party and sends $r_3 = u_1 - r_2$ to the third computing party. The second and third computing parties add the received values to their input shares. The resulting shares are $(0, u_2 + r_2, u_3 + (u_1 - r_2))$ which sum to the original value u.

The protocol in Listing 5.3 demonstrates various features of the language: the ability to perform computations and define values for only a subset of parties (lines 2 to 4), the ability to branch the computation depending on the evaluating party (lines 6 to 9), and finally the ability to receive values from certain fixed parties. The variables `r2` and `r3` are only defined by the first computing party. The result of the function body is computed differently depending on the computing party: the first computing party always returns 0 (line 7), while the second and third party add the received value to their respective input shares (lines 8 and 9).

Both resharing operations have the pattern where one of the participants generates a random value and sends it to some other party. Instead, the two parties can agree upon a common random number generator seed before the protocol is evaluated. The parties can later generate random values that would otherwise have to be sent over the network. Using this optimization, the resharing functions require no network communication. We apply this optimization automatically so that developers do not have to modify existing protocols to benefit. The details and benefits of this optimization are discussed in Section 5.4.2.

Multiplication protocol

The algorithm for multiplying two additively shared numbers $u, v \in \mathbb{Z}_{2^n}$ is based on a simple equality given by the distributivity of multiplication over addition:

$$(u_1 + u_2 + u_3)(v_1 + v_2 + v_3) = \sum_{i,j=1,1}^{3,3} u_i v_j = \sum_{i=1}^{3} (u_i v_i + u_i v_{\mathsf{p}(i)} + u_{\mathsf{p}(i)} v_i) \ ,$$

Listing 5.4: Multiplication protocol

```
def mult(u: uint[n], v: uint[n]): uint[n] = {
  let
    u = reshare(u)
    v = reshare(v)
    w = u * v + u * (v from Prev) + (u from Prev) * v;
  w
}
```

where $\mathsf{p}(i)$ denotes the previous index ($\mathsf{p}(\cdot)$ maps 2 to 1, 3 to 2 and 1 to 3). This equation is directly mapped to the code in Listing 5.4 by letting the i-th party compute the term $w_i = u_i v_i + u_i v_{\mathsf{p}(i)} + u_{\mathsf{p}(i)} v_i$. To achieve security and privacy the algorithm reshares both inputs. Notice the let-expression overshadowing input variables u and v with same names. We can see the similarity to Algorithm 2 in [23] but again, the presentation is much more concise.

Textually the resharing calls happens one after another but notice that there are no data dependencies that forbid us from performing both in parallel during the first communication round. This is exactly what happens in practice where we try to minimize the number of communication rounds. The simplest approach is to greedily send network messages in the earliest communication round possible.

Usually we specify protocols for integers of arbitrary bit width n, like the multiplication protocol here, but in a concrete system protocol implementations are instantiated to bit widths that computers support natively. In most cases protocols are specialized to operate on 8-, 16-, 32- and 64-bit integers. The only limitation is that we do not allow bit width to be chosen dynamically. It always has to be fixed before executing the code, during the compilation of protocols. Support for arbitrarily large integers has turned out to be extremely useful. For example, the integer division protocol internally uses integers larger than 128 bits. In addition to that, as we will later see, fixed- and floating-point computations can be sped up by starting operations on large integers and gradually cutting back during the protocol.

Disregarding the resharing calls, the multiplication protocol that we have just implemented is not symmetric in communication. When resharing does not perform network communication the multiplication protocol sends both messages `(v from Prev)` and `(u from Prev)` over the same network channel. One-sided communication does not benefit from the full-duplex nature of Ethernet connections. We have proposed [64] a small modification to the protocol that achieves balanced communication, namely the result is $u_i v_i + u_{\mathsf{p}(i)} v_i + u_{\mathsf{p}(i)} v_{\mathsf{n}(i)}$. As previously, $u_{\mathsf{p}(i)}$ is computed as `(u from Prev)` and $v_{\mathsf{n}(i)}$ as `(v from Next)`. While the expression for sending u will occur in the implementation twice, the data is only sent once. We showed [64] that this optimization results in a small speedup of around 1.1.

Listing 5.5: Basic bit-level protocol

```
def xorReshare(u: uint[n]): uint[n] = {
  let r = rng();
  u ^ r ^ (r from Next)
}
def conj(u: uint[n], v: uint[n]): uint[n] = {
  let
    u = xorReshare(u)
    v = xorReshare(v);
  u & v ^ u & (v from Prev) ^ (u from Prev) & v
}
def disj : uint[n] -> uint[n] -> uint[n] = \u v -> ~conj(~u, ~v)
```

5.1.3 Bit-level operations

Many of the high-level protocols are implemented in terms of bit-level operations on values secret shared over $\mathbb{Z}_2^n$. To see how we can implement basic bit-level operations, consider the special case of additive secret sharing over the ring $\mathbb{Z}_2$. Multiplication over $\mathbb{Z}_2$ acts as Boolean conjunction $\wedge$ and addition as exclusive OR $\oplus$. This yields a simple recipe for implementing bitwise additive sharing over $\mathbb{Z}_2^n$. XOR is implemented by locally computing the XOR of respective shares. The bitwise conjunction protocol (Listing 5.5) is almost identical to additive multiplication but addition is replaced by XOR and multiplication of shares by bitwise conjunction. Bitwise negation is computed by an odd number of parties negating their shares, and disjunction can be computed via conjunction and negation via De Morgan's laws. We call this kind of bitwise additive sharing *XOR sharing*.

Prefix-or

Prefix-or is a primitive bit-level operation that is often used inside higher-level protocols. It is excellent for demonstrating recursion and size-polymorphic functions. The prefix-or of a value $\vec{u} \in \mathbb{Z}_2^n$ is obtained by propagating its most significant 1 bit downwards. For example, the prefix-or of the 8-bit number 00101100_2 is 00111111_2. If $\vec{u}_i$ denotes the i-th bit of $\vec{u}$, then the prefix-or of $\vec{u}$ is $\vec{v}$ where $\vec{v}_i = \bigvee_{j=i}^{n} \vec{u}_i$.

The implementation of prefix-or is shown in Listing 5.6. The `prefixOr` protocol is defined recursively: we split the input into two roughly equally long parts and recursively compute the prefix-or of the parts. If the upper half has any 1 bits then all lower half bits will also need to be set. To achieve this, we compute the disjunction of the resulting lower bits with the lowest bit of the upper half. Finally, we concatenate the resulting parts. For 0- or 1-bit number we return the number

Listing 5.6: Prefix-or protocol

```
def prefixOr(p: uint[n]): uint[n] = {
  if (n < 2) return p;
  let
    x = prefixOr(p[0 .. n/2])
    y = prefixOr(p[n/2 .. n])
    x = disj(x, lift(y[0]));
  x ++ y
}
```

itself; this is the recursion base case.

This example demonstrates multiple features of the language. First, the function is defined recursively. Second, integers can be manipulated as arrays. In fact, integer type `uint[s]` is actually a synonym for bit array `arr[bit,s]`. The syntax `u[m..n]` denotes the slice of the array `u` from index `m` until the index `n-1`.

The example uses few operations that we have not defined previously. The operator `++` is defined in the standard library and concatenates two arrays. The built-in function `lift` of type `a -> arr[a,n]` takes a value of any type and replicates it to desired length. The number of duplicates depends on the context and is derived from the return type. Concretely, the expression `lift(y[0])` takes the least-significant bit of `y` and replicates it to n-bit integer (bit array).

Share conversion from XOR to additive

Bit-level protocols are used as building blocks for high-level additive protocols. To use these building blocks we need to convert between additive sharing and XOR sharing. Converting a bit $u \in \mathbb{Z}_2$ to an additively shared integer in $\mathbb{Z}_{2^n}$ is not completely straightforward, because the shares of u may also add up to 2 or 3 in $\mathbb{Z}_{2^n}$. The implementation of this protocol called `shareconv` (Listing 5.7 adapted from [23]) is quite involved but only performs simple operations. The first party generates a random bit $b \in \mathbb{Z}_2$ and reshares $m = b \oplus u_1 \in \mathbb{Z}_{2^n}$, where u_1 is the first party's share of u, between the second and the third party. Next, the second and third party compute $s = b \oplus u_2 \oplus u_3 \in \mathbb{Z}_2$. Finally, if $s = 1$ then the result is $1 - m$ and otherwise it is m.

We have restricted `shareconv` function to only return integers that have at least one bit. This is achieved via the constraint `n>0`. Whenever the share conversion function is called, the type checker verifies that its result has at least one bit.

Using the share conversion protocol we can convert an arbitrary bit width XOR-shared integer $\vec{u} \in \mathbb{Z}_2^n$ to an additively shared integer by converting each bit $\vec{u}_i$ to $v_i \in \mathbb{Z}_{2^n}$ and then computing the dot product with public successive powers

Listing 5.7: Share conversion

```
def shareconv[n>0](u: uint[1]): uint[n] = {
  if (n == 1) return u;
  let {1}
    b12 = rng ()
    b13 = rng ()
    b = b12 ^ b13
    m = zextend (b ^ u)
    m12 = rng ()
    m13 = m - m12;
  let {2} s23 = (b12 from 1) ^ u;
  let {3} s32 = (b13 from 1) ^ u;
  let {2,3} s = (s23 from 2) ^ (s32 from 3);
  party:
    1 -> 0
    2 -> if (s == 1) (1 - (m12 from 1)) else (m12 from 1)
    3 -> if (s == 1) (0 - (m13 from 1)) else (m13 from 1)
}
```

Listing 5.8: XOR to additive conversion

```
def xorToAdditive(bits: uint[n]): uint[n] = {
  if (n == 0) return 0;
  sum(zipWith(\b i -> shareconv(arr{b}) << i, bits, countUp(0)))
}
```

of two: $\sum_{i=0}^{n-1} 2^i v_i$. We call this function `xorToAdditive`. The implementation (Listing 5.8) is straightforward using some standard library functions. The function `countUp` takes an integer and produces an array of successive integers starting from the input value. The length of the result is derived from the output type **arr**`[`**uint**`[n],k]` where **uint**`[n]` is the input type. Higher-order function `zipWith` takes a function of two arguments and two arrays of same length and applies the function to the input arrays pointwise. Expression **arr**`{b}` converts the bit `b` into an integer of length one. We need to take special care of 0-bit integers because the share conversion function can not return 0-bit integers.

Additive type conversions

The problem of converting a single bit to a larger additive representation can be generalized to the question if the sum of shares *overflows*. This same problem occurs in the additive scheme if we want to convert any n-bit number to a larger $(n+m)$-bit number while keeping the original value. The solution is to implement

Listing 5.9: Overflow protocol

```
def msnzb(u: uint[n]): uint[n] = {
  if (n < 2) return u;
  let u = prefixOr (u);
  u ^ (u >> 1)
}
def mszb(u: uint[n]): uint[n] = msnzb(~u)
def overflow(x: uint[n]): uint[1] =
  xorBits(conj(mszb(x), party: 1->0 2->x 3->0))
```

a general `overflow` function that returns one if the sum of shares overflows. Note that a sum of three number may overflow two bits. Therefore, `overflow` should only be used on values that are shared between two parties.

The `overflow` function is defined in Figure 5.9. It operates by first finding the most significant positions where the input bits are equal and then checks if those bits are either 1 or 0. If those bits are both 1 then they generate a carry and subsequent bits will propagate that carry over the last bit. If the highest equal bits are both 0 then they cancel any incoming carry and higher bits can not generate a carry either.

The helper function `mszb` takes a XOR shared number and finds its highest 0 bit. It is defined through `prefixOr`. If the highest zero bit of `x` is at index i then `mszb(x)` return XOR shared 2^i. If the input does not have 0 bits then `mszb` returns 0. We have also used a standard library function `xorBits` that takes an integer and finds the XOR of its bits.

Using this primitive we can implement many useful functions: `cut` to eliminate some lower bits of a number, `extend` to convert a number to one with more bits, and `publicShiftRight` to shift the input right by some public number of bits. The functions are implemented in Listing 5.10. Notice how the `cut` function differs from bit shift in that it explicitly removes lower bits. This is an optimization over `publicShiftRight` that has to compute both upper and lower overflow bits—`cut` only has to compute lower overflow bits. This is usually a considerable speedup.

We use the syntax `` `s `` to convert a size type `s` to an integer value. The backtick operator accepts any size expressions. For example `` `(n+m) `` is also a valid expression. In the current example this type to value conversion is needed because primitive bit shift operations expect integer arguments. Notice also that `extend` function has type **`uint`**`[n] ->` **`uint`**`[m+n]` expressing that the input can be extended by any number of bits m. When we write `` extend`[m = 8](x) `` we extend the input `x` by 8 bits. The variable `m` to the left of the equals sign indicates the type argument `m` of the `extend` function.

Listing 5.10: Additive type conversion protocols

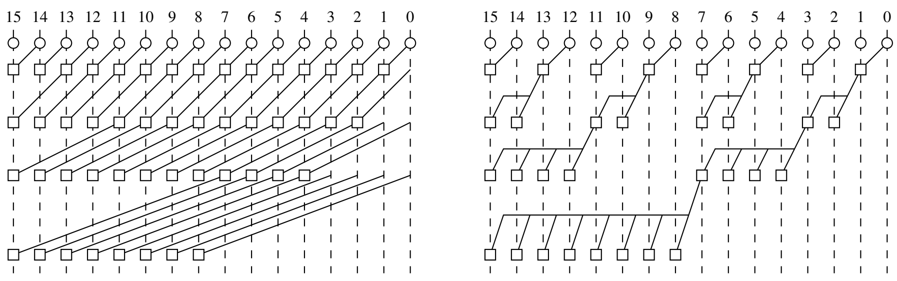
```
def extend(u : uint[n]) : uint[m + n] = {
  if (n == 0) return 0;
  if (m == 0) return u;
  let u = reshareToTwo(u);
  zextend(u) - (shareconv(overflow(u)) << `n)
}
def cut(u : uint[n + m]) : uint[n] = {
  if (n == 0) return 0;
  if (m == 0) return u;
  let u = reshareToTwo(u);
  u[m ..] + shareconv(overflow(u[.. m]))
}
def publicShiftRight[n > 0](u : uint[n], p : uint[n]) : uint[n] = {
  let u = reshareToTwo(u)
      s = u << (`n-p)
      o1 = shareconv(overflow(u))
      o2 = shareconv(overflow(s));
  (u >> p) - (o1 << (`n-p)) + o2
}
```

Additive to XOR share conversion

We have seen how to convert XOR shared data to additively shared data but the inverse is also required. Our protocol relies on computing the sum of two XOR-shared integers using primitive operations such as conjunction, disjunction and XOR. Therefore, before we present our conversion protocol we first need to discuss adder circuits. Our task is to add two n-bit integers $A = [A_1 A_2 \dots A_n]$ and $B = [B_1 B_2 \dots B_n]$ where $A_i, B_i \in \mathbb{Z}_2$ and A_1 is the least-significant bit of A. We want to compute the bits of $A + B$ efficiently.

We denote the sum of A and B as $S \in \mathbb{Z}_2^n$ and its bits as S_i. Let C_i denote whether the addition of A and B generated a carry to the i-th bit. In other words, C_i indicates if the addition of $(i-1)$-bit integers $A[1 : i-1]$ and $B[1 : i-1]$ overflows. It is easy to see that from carry bits we can compute the sum as $S_i = A_i + B_i + C_{i-1}$. Here we will assume that the incoming carry C_1 is zero. A carry bit C_{i+1} is set if at least two of the three bits A_i, B_i and C_i are set. This can be expressed as $C_{i+1} = A_i B_i + C_i A_i + C_i B_i$ where bit operations are performed in $\mathbb{Z}_2$. Alternatively, we can write $C_{i+1} = G_i + C_i P_i$ where *generator* $G_i = A_i B_i$ and *propagator* $P_i = A_i + B_i$. Using that terminology $S_i = P_i + C_{i-1}$.

Let us define a new binary operator $(P, G) \boxtimes (P', G') = (PP', G + G'P)$ in $\mathbb{Z}_2 \times \mathbb{Z}_2$. It is easy to see that this is a closed and associative (but not commutative) operator. Let $T_i = (P_i, G_i)$ and $Q_1 = T_1$ and $Q_{i+1} = T_{i+1} \boxtimes Q_i$.



(a) Kogge-Stone (b) Ladner-Fischer

Figure 5.2: Parallel prefix-sum circuit constructions

Proposition 4. *If* $C_1 = 0$ *then* $Q_i = (P_i \dots P_2 P_1, C_{i+1})$.

Proof. The proof is straightforward by induction. First, $Q_1 = T_1 = (P_1, G_1)$ and $C_2 = G_1 + C_1 P_1$. Because $C_1 = 0$ we have $G_1 = C_2$. For the induction step assume that $Q_i = (P_i \dots P_2 P_1, C_{i+1})$. By definition

$$\begin{aligned} Q_{i+1} &= T_{i+1} \boxtimes Q_i = (P_{i+1}, G_{i+1}) \boxtimes Q_i \\ &= (P_{i+1} P_i \dots P_2 P_1, G_{i+1} + C_{i+1} P_{i+1}) \\ &= (P_{i+1} P_i \dots P_2 P_1, C_{i+2}) \ . \end{aligned}$$

□

From Q_i it is efficient to recover the carry bit C_i and, therefore, also the sum bit S_i. Notice that because $\boxtimes$ is associative we do not have to compute the prefix-sum of T_i in the exact same manner as we saw in the proof. The simple approach has a large *delay* that would translate to a large round count in the implementation. Fortunately, it is well known that the prefix-sum of any associative operator can be computed in a logarithmic number of parallel operations. Two well-known parallel prefix-sum schemes [67, 72] are depicted in Figure 5.2.

Equipped with the knowledge about adder circuits, the protocol implementation is very simple. Given an additively shared value $v \in \mathbb{Z}_{2^n}$ we first reshare it between the second and the third party as $(0, v_2, v_3)$ such that $v = v_2 + v_3$. The reshared value can then be viewed as two XOR-shared values $x = (0, v_2, 0)$ and $y = (0, 0, v_3)$. Unfortunately, if we simply add them in $\mathbb{Z}_{2^n}$ we will get an additively shared sum. If we implement an adder circuit that uses conjunction and XOR, we get properly XOR-shared result. Bit extraction is defined in Listing 5.11.

The addition circuit can be implemented in a similar recursive style as the prefix-or in Listing 5.6 but we opt for a different approach. A serious problem with the circuit construction scheme used in prefix-or is that it is quite heavy in local computations. Protocols eventually get translated into straight line LLVM code that

Listing 5.11: Bit extraction

```
def bitextr(v: uint[n]): uint[n] = {
  if (n <= 1) return v;
  let
    v = reshareToTwo(v) // (0, v_2, v_3)
    x = party: 1->0 2->v 3->0 // (0, v_2, 0)
    y = party: 1->0 2->0 3->v; // (0, 0, v_3)
  xorAdd(x,y)
}
```

does not contain loops or functions. Therefore, every computation step has to occur as an explicit instruction in the generated code. Large amounts of code stress the tools that we use and degrade the performance of protocols. Currently, the only workaround for this issue is to write code that avoids manipulating individual bits and, instead, operates in SIMD fashion.

The prefix-or protocol computes prefix disjunction using the generic prefix-sum by Ladner and Fischer [72] in a bottom-up manner. The circuit for 16-bit case is depicted in Figure 5.2b. The implementation is actually communication-efficient but does not seem to admit to a good software implementation. Instead, we implement XOR shared addition using the Kogge-Stone [67] construct (Figure 5.2a) which is easier to implement computation-efficiently in software. As bit extraction is a very commonly used primitive we get much smaller protocol binaries with a reduced local computation overhead. We found that for 64-bit prefix-or, Kogge-Stone takes about 1.5 times more communication but results in a DAG that is about 20 times smaller.

Addition of XOR shared numbers is implemented in Listing 5.12. The `xorAdd` function is restricted to operate on integers at least two bits long. The function first defines the initial generator `g` and the initial propagator `p` as we previously discussed. Next, it calls a recursively defined helper function with the type parameter `k` set to 1. The helper function computes a single layer of the circuit (see Figure 5.2a) starting from the top. In the helper function we use the notation `` `k `` to convert the size type `k` to its integer value. Notice that the recursive call `` xorAddLoop`[k = 2*k](...) `` is made multiplying the type arguments by two. We stop the recursion when shifting by `k` would shift out all bits of the input.

5.1.4 Building high-level protocols

High-level protocols are usually defined by composing simpler protocols and using higher-order functions to facilitate the composition. For example, it is often useful to apply a protocol to every element of an array.

Listing 5.12: XOR shared addition protocol

```
def xorAdd[n>1](x: uint[n], y: uint[n]): uint[n] = {
  let
    g = conj(x, y)
    p = x ^ y
    g' = xorAddLoop`[k=1](g, p, 1);
  p ^ g' << 1
}
def xorAddLoop[k](g: uint[n], p: uint[n], m: uint[n]): uint[n] = {
  if (n <= k) return g;
  let
    g' = conj(p, g << `k)
    p' = conj(p, (p << `k) ^ m);
  xorAddLoop`[k = 2 * k](g ^ g', p', m ^ (m << `k))
}
```

As an example we will define the private right-shift protocol. It uses the following functions and protocols that we have not previously seen.

- The built-in higher-order map operation takes a function of type `a -> b` and applies it to the input array **arr**[a,n] pointwise. This function can actually be defined in the language but is currently built in for a more efficient mapping over integer values.
- The standard library function `xor` takes an array of integers and aggregates them with XOR.
- The built-in function `lift` takes a value of type `a` and replicates it `k` times to produce an array of type **arr**[a,k]. Thanks to type inference the number of repetitions `k` can be learned from the result type.
- The built-in function `trunc` takes an array of length n and returns an array of length m that contains only the first m elements of the input array. In the case of integers (bit arrays) the function drops most-significant bits. The function requires that m is smaller than n.
- Recall that the standard-library function `countUp` takes an n-bit integer x and returns an array that contains x at the first position, $x + 1$ at the second position, and so on.

In addition to the higher-level building blocks, right shift also uses a low-level protocol that takes an additively shared integer and returns an XOR shared m-bit integer where the i-th lowest bit indicates if the input was equal to i. In other words, the `chVector` [77] protocol computes the *characteristic vector* of the input. The

Listing 5.13: Characteristic vector protocol

```
1 def chVector[l,n>0,m>=n](v: uint[n]): uint[m] = {
2   let
3     v = reshare(v)
4     c : uint[m + l] = xorReshareToTwo(rotate(share(1), v));
5   trunc (party:
6     1 -> 0
7     _ -> rotate(c, v from 2 + v from 3))
8 }
```

Listing 5.14: Additive bit shift right protocol

```
1 def shiftr[8 <= n](u: uint[n], s: uint[8]): uint[n] = {
2   let
3     u = bitextr(u)
4     vs = map(\i -> u >> i, countUp(0))
5     bs : uint[n] = chVector`[l = 0](s)
6     rs = zipWith(\v b -> conj(v, lift(b)), vs, bs);
7   xorToAdditive(xor(rs))
8 }
```

protocol implemented in Listing 5.13 takes an additional type parameter l such that $m + l$ is a power of two. This is because the protocol only works correctly for results that are a power of two bits long, and the language currently lacks the power to express the constraint that a size type is a power of two, or to perform the rounding to the next power of two automatically.

The protocol itself operates the following way. The first party constructs the characteristic vector c from the share v_1 and XOR-reshares the result between the other parties (line 4). The second and third party then rotate c by the sum of v_2 and v_3 (line 7). The implementation makes use of a local `rotate` function that rotates the first argument right by the amount specified by the second argument.

The right-shift protocol itself is defined in Listing 5.14. It takes an additively shared integer and an additively shared shift, and returns the shifted value. The shift amount is strictly required to be 8 bits long and the input must be at least that long. The implementation is quite straightforward and compact. We first convert the input to XOR shared form and compute all possible n shifts (lines 3 and 4). Next, we build an n-element bit vector where the i-th bit indicates whether the shift is by i bits (line 5). Finally, we mask every shift with the respective bit and XOR the results (line 6). This yields a XOR shared result that we convert to additive form (line 7).

Listing 5.15: Floating-point number representation

```
struct float[m,n] = {
    sign: uint[1]
    exp : uint[m]
    frac: uint[n]
}

def floatMinus(x: float[m, n]): float[m, n] =
  float{~x.sign, x.exp, x.frac}
```

5.1.5 Floating-point operations

We have used the protocol DSL to implement floating-point arithmetic and most of the primitive operations from [62]. A floating-point number N is composed of three parts: sign bit $s \in \mathbb{Z}_2$, n-bit significand (fractional part) f, and m-bit exponent e such that $N = (-1)^s \cdot f \cdot 2^{e-b}$. The highest bit of the significand is always 1 and the exponent is biased by b. The value of bias b depends on the number of bits of the exponent. For example, it could be chosen to be $b = 2^{m-1} - 1$. The significand is represented as an n-bit value but in Sharemind the highest bit interpreted to have the value of $1/2$, the second highest $1/4$, and so on Because the highest bit must always be set, the mantissa is in the range $[1/2, 1)$.

In our implementation we have $n = 32$ for single-precision and $n = 64$ for double-precision floating-point numbers. The length of the exponent is 16 bits for both formats, but the number of bits actually used for storing the exponent is 8 bits for single-precision and 11 bits for double-precision numbers. This is to ensure backwards compatibility with the previous implementation. In addition to providing different accuracy guarantees, our floating-point numbers do not match the IEEE standard because they do not include some special values such as infinities and the not-a-number value. The mismatch is in order to simplify the implementation and provide better performance.

In the protocol DSL floating-point numbers can be represented using data structures. In Listing 5.15 we declare a new structure with fields for sign, exponent, and fractional part. The structure is generic over the width of the exponent and fractional part. The number is represented in a generic way because usually the operations are defined in a similar manner for 32- and 64-bit floating-point numbers, and this allows to define an operation for both bit widths by writing just a single polymorphic function. As an example of how structures are used, we define the trivial negation function `floatMinus` that simply inverts the sign bit.

As another example we demonstrate a protocol for computing the inverse of a floating-point number. The intuitive idea is that for $N = (-1)^s \cdot f \cdot 2^{e-b}$ we have

$$\frac{1}{N} = (-1)^s \frac{1}{f 2^{e-b}} = (-1)^s \frac{1}{2f} \frac{1}{2^{(e-b)-1}} = (-1)^s \cdot \frac{1}{2f} \cdot 2^{(1-e+2b)-b} \quad .$$

This results in a suitable floating-point representation for every $f \in (1/2, 1)$ because then $\frac{1}{2f} \in (1/2, 1)$. When the fractional part of the input is close to or exactly $1/2$ the algorithm is self-correcting, as rounding the intermediate results down prevents it from overflowing. This gives us the recipe for computing the inverse of a floating-point number. We first compute $1 - f$ and interpret it as a fixed-point number with a single binary digit before the radix point. To do that we divide $-f$ by 2 (negation and division are computed as for an unsigned integer). The fixed-point format with a single binary digit before the radix point allows us to represent values in the range $[0, 2)$ and the extra digit is needed because inverse yields us a value in the range $[1, 2)$.

By setting $x = 1 - f$ we can compute $1/f$ using the equality

$$1/f = 1/(1-x) = \sum_{i=0}^{\infty} x^i = \prod_{i=0}^{\infty} \left(x^{2^i} + 1 \right) \quad .$$

Evaluating just the first k terms of the product gives the maximum error of about 2^{-2^k} at $x = 1/2$. This means that for a single-precision floating-point number it is sufficient to only compute the first 5 terms. To get the fractional part of the result all that is left to do is to evaluate that expression on fixed-point numbers. Note that this approximates $1/f$ as a fixed-point number with one digit before the radix point, but reinterpreting it as having no digits before the radix point yields the approximation for $1/(2f)$.

To find $1/f$ we need to compute powers of $x = 1 - f$ and multiply the terms incremented by one to approximate the desired value. Therefore, fixed-point multiplication is needed. Let u and v be $(1 + n)$-bit fixed-point numbers with a single digit before the radix point. To compute the product $u * v$ (assuming that the result does not overflow) we extend both numbers to $1 + 2n$ bits, multiply them, and then cut away the least significant n digits of the result. In additive secret sharing this is a rather expensive operation: to extend the numbers we need to compute their overflow bits. To cut away least the significant digits we again need to check if those digits overflow.

If we know ahead of time that we have to perform several fixed-point multiplications in a row, we can optimize the computation by eliminating the need to extend the numbers before every multiplication. If we know that we are going to perform exactly r multiplications on u we can immediately extend. u to $1 + (1 + r)n$ bits and on every successive multiplication remove the lowest n bits.

Listing 5.16: Floating-point reciprocal protocol

```
def bias[m > 3](): uint[m] = (1 << `(m - 1)) - 1

def floatInv[r > 1, n > m, m > 3](N: float[m,n]): float[m,n] = {
  let
    x = publicBitShiftRight(-N.frac, 1) // x = 1 - f
    f' = fixInv`[r=r](x)
    e' = share((bias() + 1) << 1) - N.exp // 2 - e
    // b indicates if 1/(2f) < 1/2
    b : uint[1] = cut(f')
    half = share(1 << `(n - 1))
    f' = choice(b, f', half) // correct fraction
    e' = e' - shareconv(b); // correct exponent
  float {N.sign, e', f'}
}

def fixInv[r > 1, n > 0](x: uint[n]): uint[n] = {
  let
    x : uint[n + r*(n - 1)] = extend(x)
    one = share(1 << `(n - 1));
  fixInvLoop`[r = r - 2](x[.. n + (r - 1)*(n - 1)] + one, x)
}

def fixInvLoop[n > 0](
       acc: uint[n+(r+1)*(n-1)],
       xPow: uint[n+(r+2)*(n-1)]): uint[n] = {
  let
    xPow = cut(square(xPow))
    one = 1 << `(n - 1)
    acc : uint[n+r*(n-1)] = cut(mult(acc, xPow + one));
  if (r == 0) acc else fixInvLoop`[r = r - 1](acc, xPow)
}
```

We have used two helper functions: `choice` for obliviously choosing between two additively shared integers, and `share` for sharing a public value so that two parties pick 0 as their shares.

The function `bias` returns the bias for m-bit exponents, the function `fixInv` computes the inverse of an n-bit fixed-point number with a single digit before the radix point and finally `floatInv` computes the inverse of a floating-point number. Both inverse functions take the number of iteration, denoted with `r`, as a type argument.

If the input has a fractional part very close to 1 then $1/(2f)$ is very close to $1/2$. During computation this may be rounded down and the highest bit can become 0, resulting in a denormalized float. To avoid this, we need to check the highest bit of the to-be significand—we denote it with b. The highest bit b is computed by cutting away all but the highest bit of the initial fractional part. If the highest bit of the to-be significand turns out to be 0 then we know that the input had a fractional part very close to 1 and the result was rounded down too much during the computation. In this case we correct both the resulting significand and the exponent.

Initially we implemented the inverse protocol using a fixed-point polynomial evaluation technique [62]. However, the protocol DSL enabled us to rapidly try out different implementations and optimizations, and we quickly found out that the approach presented here is superior to polynomial evaluation both in speed and precision. Implementing the protocol in an optimized manner in our C++ framework would have been a major undertaking.

5.2 Implementation and integration with SHAREMIND

The compiler is implemented in Haskell. The front-end performs lexical analysis, syntactic analysis, static checking, and translation to the low-level intermediate DAG representation (IR). The IR is optimized, statically checked for security and compiled to LLVM [73] code. The LLVM code can in turn be compiled and linked with SHAREMIND.

First we verify that the high-level protocol language is properly typed. This includes both data type verification and party type verification. After static checks we translate the high-level code to much simpler intermediate code that is based on the core language formalization from Section 5.3. This representation is evaluated to a normal form which is converted to the IR. The IR has a well-defined syntax and semantics and is optimized with a separate tool. Security analysis is performed on the IR to provide additional guarantees that optimizations preserve security.

The well-specified IR facilitates various tools to make protocol development easier. For example, in addition to the optimizer and security checker we also have implemented a tool to directly evaluate IR programs. The evaluator drastically simplifies and accelerates protocol development: new protocols can be tested without having to integrate them with SHAREMIND and having to deal with the complicated networked application. Only when the developer is sure that a new protocol is correct, secure and performs reasonably well can it finally be integrated with the platform.

It is possible to estimate performance of protocols by only analyzing the generated IR code. We found (see Section 5.6.5) that the lowest performance in operations per second can be estimated very accurately as a linear function of round count. The highest achieved performance can be estimated as a linear function of communication cost.

The compiler generated code is not actually tightly coupled to SHAREMIND and does not depend at all on Sharemind's functionality. Instead control is inverted so that the generated code provides meta-information and various callbacks. The callbacks need to be invoked with requested data such as received messages and randomly generated bits. Other SMC systems can use the generated code, given that they are able to send network messages and provide random numbers.

5.3 Language semantics

In this section we will formalize the type system and dynamic behavior of the core of the protocol DSL. We will look at the data type system and party type system separately. The core language is not as syntactically rich as the high-level one and a major difference is that type arguments are not inferred automatically. The core language can be considered a mid-level intermediate language in our compilation pipeline. For some constructs we provide syntactic rewriting rules from the high-level language to the low-level one.

5.3.1 The core language

The syntax of the core language is presented in Figure 5.3. Expressions include the standard constructs for lambda calculus: variables, function applications, lambda abstractions and let-expressions. The language also includes conditional expressions over type-level size predicates, case-expressions for branching depending on the computing party, and from-expressions for performing network communication. The language lacks general type abstraction expressions as type arguments are implicitly introduced at the top-level. The language supports restricting functions to some type constraints C. The constraints may contain conjunctions, negations, size type equalities and less-than relations.

Expression are always evaluated by a set of parties that may communicate with each other. By default, all parties evaluate the same expression, but every computing party does not always hold a result for the given expression. Case-expressions allow the computation to branch depending on the evaluating party but may also omit the resulting value for some parties. Let-expressions are not recursive and shadow (override) previous variable definitions with the same name.

From-expression e `from` $\langle p_1, \ldots, p_n \rangle$ is used for network communication and states that the i-th evaluating party gets the value of the expression e from party p_i. For example, the expression x `from` $\langle 1, 1, 1 \rangle$ evaluates to the first computing party's value of the variable x for all computing parties (including the first one). The high-level language contains metavariables `Next` and `Prev` that can be rewritten as $\langle 2, 3, \ldots, n, 1 \rangle$ and $\langle n, 1, 2, \ldots, n-1 \rangle$ respectively.

Conditional expression `if` C `then` e_1 `else` e_2 evaluates to e_1 if type constraint

$$\begin{array}{llcl}
\text{Size literals} & c & \in & \mathbb{N}_0 \\
\text{Party no.} & p & \in & \mathbb{N}_1 \\
\text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \mid e\, \tau \\
 & & \mid & \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \\
 & & \mid & \texttt{if}\, C\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \\
 & & \mid & \texttt{case:}\, p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n \\
 & & \mid & e\, \texttt{from}\, \langle p_1, \ldots, p_n \rangle \\
\text{Programs} & M & ::= & \epsilon \mid \texttt{def}\, x : \sigma \; = \; e\, M \\
\text{Constraints} & C & ::= & \varepsilon \mid C_1 \wedge C_2 \mid \neg C \mid \tau_1 \sim \tau_2 \mid s_1 < s_2 \\
\text{Kinds} & k & ::= & \texttt{type} \mid \texttt{size} \\
\text{Monotypes} & \tau, s & ::= & \alpha \mid \texttt{unit} \mid \texttt{bit} \mid \texttt{arr[}\tau\texttt{,}\ s\texttt{]} \mid \tau_1 \rightarrow \tau_2 \\
\text{Size types} & & \mid & c \mid s_1 + s_2 \mid s_1 * s_2 \mid s_1 / s_2 \\
\text{Polytypes} & \sigma & ::= & \forall(\alpha : k).\, \sigma \mid C \Rightarrow \tau
\end{array}$$

Figure 5.3: Syntax of the core of the protocol DSL

C holds and to e_2 otherwise. When type checking the first branch the fact that C holds may be used. Respectively, when type checking the second branch the fact that C does not hold is used. The protocol DSL compiles to an intermediate representation (IR) with no branching constructs, meaning that source code may only contain loops that are statically bounded. The combination of supporting type-level integers and providing the ability to branch over them facilitates writing recursive code and cleanly segregates values that might only be dynamically known (regular values) from values that are definitely statically known (types).

Unlike low-level language the high-level language also supports branching over runtime values. Same functionality can be implemented using a function that takes the conditional and both branches as arguments and chooses the correct results based on the conditional. Note that we do not currently have static program termination checks and well-typed programs are not guaranteed to terminate. For example, a recursive call inside an if-expression over a dynamic value will result in an infinite loop during translation to intermediate form. We have not found this to be a problem in practice.

Case-expression $\texttt{case:}\ p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n$ evaluates to e_i for party p_i. This construct is slightly different from what we have seen in the previous examples but high-level code can be straightforwardly translated to this form. A case over a set of parties P can be implemented by binding the expression e to a fresh variable t, replacing the expression with the variable, and duplicating the resulting branch for each party $p \in P$.

A program M consists of a sequence of variable definitions. All top-level bindings must be annotated with types. Unlike let-expressions, top-level definitions may be mutually recursive. In the core language we are not modeling the protocol definitions of the high-level language. Monomorphic, first-order functions can be considered to fill the role of concrete protocols.

5.3.2 Data type system

The type system of the language is inspired by Cryptol [83]. We have opted for strict and static type checking with type inference: a classic Hindley-Milner type system [34] extended with type constraints (predicates) over type-level natural numbers. To simplify type inference we require top-level definitions to have explicit types and we do not support polymorphic let-expressions.

The syntax of types is presented in Figure 5.3. A regular type τ is either a type variable α, the `unit` type having only a single value, the `bit` type having two values 0 and 1, an array $\texttt{arr[}\tau\texttt{, }s\texttt{]}$ of length s containing elements of type τ, or a function $\tau_1 \to \tau_2$ taking arguments of type τ_1 and returning values of type τ_2. Because most protocols operate on integer values we use $\texttt{uint[}n\texttt{]}$ as a synonym for an array of n bits for the sake of conciseness and readability. A size type s is either a variable α, a natural number $c \in \mathbb{N}_0$, or an arithmetic expression of size types.

The protocol DSL has two different kinds of types: regular data types, and size types for denoting lengths of arrays. Kinds are denoted with k. Rules for checking that types are well-formed and well-kinded are presented in Figure 5.4. Judgement $\Gamma \vdash \tau : k$ denotes that type τ has the kind k under the type environment Γ.

Type constraints C are used to restrict global functions using equalities and inequalities between size types. More complicated constraints can be formed with conjunctions $C_1 \wedge C_2$ and negations $\neg C$. Type constraints can also be used for branching in if-expressions.

The type system of the DSL is provided in Figure 5.5. Judgements of the form $C; \Gamma \vdash e : \tau$ denote that expression e has type τ under type environment Γ assuming that constraint C holds. Judgement $\Gamma \vdash M$ denotes that program M is well-typed in environment Γ. Judgement $\Gamma \vdash f : \sigma = e$ denotes that the polymorphic function definition is well-typed under Γ. The entailment relation $C \Vdash D$ denotes that whenever C holds then D must hold. This is under all possible valuations of their free variables. Equalities between data types are structural and between size types standard arithmetic rules apply. The type environment maps variables to their types but also type variables to their kinds.

The typing rules are fairly standard and follow the framework from [112]. However, there are a few things to note. First, let-expressions are not generalized and polymorphic types can only be introduced in the global scope with explicit type annotation. This is to avoid having to infer size constraints automatically. Second, if-expressions do not allow for free type variables to occur in the size predicate. Finally, global declarations in a program are all potentially mutually recursive;

$$\frac{(\alpha : k) \in \Gamma}{\Gamma \vdash \alpha : k}$$

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{type}} \quad \frac{}{\Gamma \vdash \texttt{bit} : \texttt{type}} \quad \frac{\Gamma \vdash \tau : \texttt{type} \quad \Gamma \vdash s : \texttt{size}}{\Gamma \vdash \texttt{arr[}\tau\texttt{,}\ s\texttt{]} : \texttt{type}}$$

$$\frac{}{\Gamma \vdash c : \texttt{size}} \quad \frac{\Gamma \vdash s_1 : \texttt{size} \quad \Gamma \vdash s_2 : \texttt{size} \quad \oplus \in \{+, *, /\}}{\Gamma \vdash s_1 \oplus s_2 : \texttt{size}}$$

$$\frac{}{\Gamma \vdash \varepsilon} \quad \frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash C_1 \wedge C_2} \quad \frac{\Gamma \vdash C}{\Gamma \vdash \neg C} \quad \frac{\Gamma \vdash s_1 : k \quad \Gamma \vdash s_2 : k}{\Gamma \vdash \tau_1 \sim \tau_2}$$

$$\frac{\Gamma \vdash s_1 : \texttt{size} \quad \Gamma \vdash s_2 : \texttt{size}}{\Gamma \vdash s_1 < s_2}$$

Figure 5.4: Kind checking rules

hence, we require that all global type bindings occur in the initial environment. We assume that all type arguments have been explicitly provided. This is not to simplify the type checking rules but to simplify the presentation of dynamic semantics. In the actual language we infer type arguments when possible. For inferring type arguments the following rule can be added:

$$\frac{C; \Gamma \vdash e : \forall \alpha : k.\, \sigma \quad \Gamma \vdash \tau : k}{C; \Gamma \vdash e : [\alpha \mapsto \tau]\sigma} \quad .$$

The type system is simply an instantiation of *OutsideIn(X)* [112] where X has been chosen to be the natural number constraint domain. The general type checking algorithm is described in [112]. Note that the type system of the language is not complete. In order to type check arbitrary programs we need to be able to solve arbitrary (non-linear) systems of equations over natural numbers. In practice this has not turned out to be a problem as almost all constraints are very simple and easily dispatched by Z3 [89] SMT solver.

5.3.3 Party type system

The goal of the party type system is to guarantee that values are always available to every evaluating party. For example, we forbid the first party from using a variable that only the second party has previously defined. Party types are represented as sets of parties. If an expression e has type P it means that e has well-defined values for each party in set P. Another requirement is that the party type system has to support polymorphism. For example, we want arithmetic operations to be polymorphic over

$$\frac{(x : \sigma) \in \Gamma}{C; \Gamma \vdash x : \sigma} \quad \frac{C; \Gamma \vdash e : \forall \alpha : k.\, \sigma \quad \Gamma \vdash \tau : k}{C; \Gamma \vdash e\, \tau : [\alpha \mapsto \tau]\sigma} \quad \frac{C; \Gamma \vdash e : D \Rightarrow \tau \quad C \Vdash D}{C; \Gamma \vdash e : \tau}$$

$$\frac{C; \Gamma \vdash e : \tau_1 \quad C \Vdash \tau_1 \sim \tau_2}{C; \Gamma \vdash e : \tau_2}$$

$$\frac{C; \Gamma, (x : \tau_1) \vdash e : \tau_2}{C; \Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \quad \frac{C; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad C; \Gamma \vdash e_2 : \tau_1}{C; \Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{C; \Gamma \vdash e_1 : \tau_1 \quad C; \Gamma, (x : \tau_1) \vdash e_2 : \tau_2}{C; \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash D \quad C \wedge D; \Gamma \vdash e_1 : \tau \quad C \wedge \neg D; \Gamma \vdash e_2 : \tau}{C; \Gamma \vdash \mathtt{if}\ D\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

$$\frac{C; \Gamma \vdash e_1 : \tau \quad \ldots \quad C; \Gamma \vdash e_k : \tau}{C; \Gamma \vdash \mathtt{case\colon}\ p_1 \to e_1, \ldots, p_k \to e_k : \tau} \quad \frac{C; \Gamma \vdash e : \tau}{C; \Gamma \vdash e\ \mathtt{from}\ \langle p_1, \ldots, p_k \rangle : \tau}$$

$$\frac{\mathit{ftv}(\Gamma) = \emptyset \quad \Gamma' = f_1 : \sigma_1, \ldots, f_n : \sigma_n \quad \forall i \in \{1, \ldots, n\}.\, \Gamma, \Gamma' \vdash f_i : \sigma_i = e_i}{\Gamma \vdash \mathtt{def}\ f_1 : \sigma_1 = e_1\ \ldots\ \mathtt{def}\ f_n : \sigma_n = e_n}$$

$$\frac{\Gamma, (\alpha : k) \vdash f : \sigma = e}{\Gamma \vdash f : (\forall \alpha : k.\, \sigma) = e} \quad \frac{\Gamma \vdash \tau : \mathtt{type} \quad \Gamma \vdash C \quad C; \Gamma \vdash e : \tau}{\Gamma \vdash f : (C \Rightarrow \tau) = e}$$

Figure 5.5: The protocol DSL data type checking rules

input parties and also allow party-polymorphic user-defined functions that do not perform network communication.

Function arguments may be constrained by availability. If an argument is used by some specific parties then the type of the function has to reflect that this argument must be defined for at least those parties, but it can be defined for more. We capture this via the *subtyping relation* over sets of parties $P_1 \geq P_2 \iff P_1 \supseteq P_2$. If an expression e has type P_2 then it also has type P_1 whenever $P_1 \subseteq P_2$. Let $\top$ indicate the set of all available parties. For example, when type checking the program for three parties, $\top = \{1, 2, 3\}$.

In this section we will present the party type checking algorithm. The algorithm operates in two phases: first, a system of constraints is generated, and then the system is resolved. If the system is successfully resolved then top-level types can be inferred. Compared to data type checking, the party type checking rules do not directly describe what it means for an expression to be well-typed. Instead

Non-function types	π	$::=$	$u \mid P \mid \pi_1 \cap \pi_2 \mid (\beta_p)_{p \in \top}$
Availability types	β	$::=$	$\pi[\![i]\!] \mid \beta_1 \wedge \beta_2 \mid 0 \mid 1$
Monotypes	η	$::=$	$\pi \mid \rho \rightarrow \eta$
Contravariant monotypes	ρ	$::=$	$u \mid \eta \rightarrow \rho$
Polytypes	ζ	$::=$	$\forall u.\, \zeta \mid \Phi \Rightarrow \eta$
Constraints	Φ	$::=$	$\Phi_1 \wedge \Phi_2 \mid u \geq P$

Figure 5.6: The syntax of party types

the relation is indirect. Expression is well-typed if a system of constraints can be generated according to the rules we provide and the system can be resolved. Direct meaning could be given as we did for data type system in Section 5.3.2. We postulate that this could be done and the algorithm could be shown to be sound but not complete as in [112]. We will start by introducing the syntactic structure of party types.

The syntax of party types

The syntax of types is given in Figure 5.6. Top-level and built-in functions have a polymorphic type ζ that is possibly restricted by constraints Φ. The constraints require type variables u to be available to at least some subset of parties P. Types η are either non-function types π or function types from contravariant type ρ to invariant type η. We consider non-function types separately in order to restrict functions from being sent over the network or being applied depending on the computing party. Function types η are constructed so that only type variables may occur in contravariant positions. This limitation greatly simplifies type inference.

Non-function types π contain exact party types P, intersection types $\pi_1 \cap \pi_2$ and tuples $(\beta_p)_{p \in \top}$ where β_p denotes if a value is available to party p. The tuple components may directly indicate availability with 0 or 1. Conjunction $\beta_1 \wedge \beta_2$ denotes that the component is available only if both β_1 and β_2 indicate availability. Finally, $\pi[\![i]\!]$ is available if π is available for the i-th party.

Notice that party type P is equivalent to tuple $(\beta_p)_{p \in \top}$ where β_p is 1 if $p \in P$ and 0 otherwise. Similarly, $\pi_1 \cap \pi_2$ can be rewritten using projections and conjunctions. While technically neither construct is necessary, they improve readability. Intersection types allow many built-in operations to have more compact types. For example, binary addition has the type $\forall u\, v.\, u \rightarrow v \rightarrow u \cap v$ whereas without intersection types it would have to be written as

$$\forall u\, v.\, u \rightarrow v \rightarrow (u[\![1]\!] \wedge v[\![1]\!], u[\![2]\!] \wedge v[\![2]\!], u[\![3]\!] \wedge v[\![3]\!]) \ .$$

Type inference is split into constraint generation and constraint resolution.

During constraint generation we emit a slightly richer set of constraints Ψ that also contains more general subtyping relations and equality constraints:

$$\Psi ::= \Psi_1 \wedge \Psi_2 \mid \pi \geq P \mid \eta \geq \rho \mid \eta = \rho \ .$$

The constraints do not contain an empty (or constant true) constraint because this can, for example, be expressed as $\{\} \geq \{\}$. During constraint resolution we will try to find the least restrictive Φ from which Ψ can be derived. The algorithm for doing so is described next.

Constraint generation

We present constraint generation (Figure 5.7) in the style of *Algorithm W* by Damas and Milner [34] using side effects for generating fresh variables. Judgements of the form $\Gamma \vdash e : \eta \leadsto \Psi$ denote that under environment Γ expression e generates type η and constraints Ψ. All rules compute the resulting constraint as the conjunction of the constraints generated by the subexpressions and some extra constraints specific to that rule.

Let us start with the rule for lambda functions. We first generate a unique type variable u and type check the body of the function e with the variable x bound to the type u. If we find that e has type η then the lambda expression has type $u \to \eta$. As we see the rules contain side effects in the form of fresh variable generation. Hence, the rules should be taken as the description of the constraint generation algorithm. A more declarative description could be given in a style similar to that of data type checking rules in Section 5.3.2.

When type checking polymorphic functions, the universally quantified variables are replaced with fresh variables and the constraints required by the polymorphic function are returned as the result. In the case of from-expression, we restrict the subexpression to a non-function type π and return a type that shuffles π appropriately using projections and tuple construction. When type checking case-expressions we make sure that the branches have non-function types and are defined for the required parties. A case-expression is defined for the union of all parties that have been branched over. In the case of if-expressions we type check both branches, and generate a fresh variable u that must be a supertype of both branches. During constraint resolution we actually find the least common supertype.

Recall that the party type system is not directly exposed to the end user and type errors are raised only if the compiler fails to infer types for all top-level expressions. This means that for top-level bindings type generalization needs to be performed.

For the party type checker a program M is just a sequence of potentially recursively defined variable definitions $f_1 = e_1, \ldots, f_n = e_n$. To type check the program M we generate fresh variables u_i and construct the global environment $\Gamma' = f_1 : u_1, \ldots, f_n : u_n$. Let built-in function types be defined in environment Γ.

$$\frac{(x : \zeta) \in \Gamma}{\Gamma \vdash x : \zeta \rightsquigarrow \varepsilon} \qquad \frac{\Gamma \vdash e : \forall \alpha.\, \zeta \rightsquigarrow \Psi \quad u \text{ fresh}}{\Gamma \vdash e : [\alpha \mapsto u]\zeta \rightsquigarrow \Psi} \qquad \frac{\Gamma \vdash e : (\Phi \Rightarrow \eta) \rightsquigarrow \Psi}{\Gamma \vdash e : \eta \rightsquigarrow \Psi \wedge \Phi}$$

$$\frac{\Gamma \vdash e : \pi \rightsquigarrow \Psi \quad \forall i \in \{1, \ldots, n\}.\, \beta_i = \pi[\![p_i]\!]}{\Gamma \vdash e\ \mathtt{from}\ \langle p_1, \ldots, p_n \rangle : (\beta_1, \ldots, \beta_n) \rightsquigarrow \Psi}$$

$$\frac{\forall i \in \{1, \ldots, n\} \quad \Gamma \vdash e_i : \pi_i \rightsquigarrow \Psi_i \quad \Psi'_i = \Psi_i \wedge (\pi_i \geq \{p_i\})}{\Gamma \vdash \mathtt{case}\colon p_1 \to e_1, \ldots, p_n \to e_n : \{p_1, \ldots, p_n\} \rightsquigarrow \bigwedge \Psi'_i}$$

$$\frac{\Gamma \vdash e_1 : \eta_1 \rightsquigarrow \Psi_1 \quad \Gamma, (x : \eta_1) \vdash e_2 : \eta_2 \rightsquigarrow \Psi_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \eta_2 \rightsquigarrow \Psi_1 \wedge \Psi_2}$$

$$\frac{\Gamma \vdash e_1 : \eta_1 \rightsquigarrow \Psi_1 \quad \Gamma \vdash e_2 : \eta_2 \rightsquigarrow \Psi_2 \quad u \text{ fresh} \quad \Psi = (\eta_1 \geq u) \wedge (\eta_2 \geq u)}{\Gamma \vdash \mathtt{if}\ C\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : u \rightsquigarrow \Psi \wedge \Psi_1 \wedge \Psi_2}$$

$$\frac{\Gamma, (x : u) \vdash e : \eta \rightsquigarrow \Psi \quad u \text{ fresh}}{\Gamma \vdash \lambda x.\, e : (u \to \eta) \rightsquigarrow \Psi}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \eta_1 \rightsquigarrow \Psi_1 \quad \Gamma \vdash e_2 : \eta_2 \rightsquigarrow \Psi_2 \\ u_1, u_2 \text{ fresh} \quad \Psi = (\eta_1 = u_1 \to u_2) \wedge (\eta_2 \geq u_1)\end{array}}{\Gamma \vdash e_1\ e_2 : u_2 \rightsquigarrow \Psi \wedge \Psi_1 \wedge \Psi_2}$$

Figure 5.7: Party type constraint generation algorithm for expressions

Next, we perform type inference for all top-level definitions $\Gamma, \Gamma' \vdash e_i : \eta_i \rightsquigarrow \Psi_i$ and attempt to resolve all generated systems of constraints Ψ_i yielding some Φ_i and substitutions θ_i that map variables to types. We need to solve all constraints at the same time as mutually recursive functions influence each other's types. Constraint resolution is described in the following section. Finally, each top-level definition f_i from the recursive block gets the type $\forall \overline{\alpha_i}.\, \Phi_i \Rightarrow \theta_i \eta_i$ where $\overline{\alpha_i}$ are the free type variables of the substituted η_i. In the actual implementation we sort the top-level bindings topologically and perform type checking a single recursive block at a time. This improves efficiency and gives type errors that refer back to the previously well-typed functions.

Constraint resolution

Constraint resolution is performed by simplifying the constraints Ψ until we either get stuck, in which case the program is ill-typed, or until we reach a suitable form. In the first phase we substitute type variables and apply the following simplification rules:

$$\begin{array}{rcl}(\rho_1 \to \eta_1) = (\eta_2 \to \rho_2) & \longrightarrow & \eta_1 = \rho_2 \wedge \eta_2 = \rho_1 \\ (\rho_1 \to \eta_1) \geq (\eta_2 \to \rho_2) & \longrightarrow & \eta_1 \geq \rho_2 \wedge \eta_2 \geq \rho_1 \\ (\rho_1 \to \eta_1) \geq u & \longrightarrow & \eta_1 \geq u_2 \wedge u_1 \geq \rho_1 \wedge u = u_1 \to u_2 \\ u \geq (\eta_2 \to \rho_2) & \longrightarrow & u_2 \geq \rho_2 \wedge \eta_2 \geq u_1 \wedge u = u_1 \to u_2 \ ,\end{array}$$

where u_1 and u_2 are fresh variables. We keep track of the substitutions that have been performed. The second rule is derived from the standard subtyping principle for functions. The last two rules are used when the subtyping relation forces a type variable to a function.

During variable substitution we must take care to perform occurs checks and take care to not invalidate the structure of types. For example, substituting a function under an intersection type will yield a type error. Initially, the system of constraints contains only type equalities only in the form $\eta = u_1 \to u_2$. If during substitution we find that η is a non-function type π, and not a type variable, we will need to raise a type error. If η is a type variable we can substitute it because the right-hand side may safely occur in both covariant and contravariant positions. If η is a functional type, we simplify it according to the first rule. During simplification we again only introduce substitutions $u = u_1 \to u_2$ that are safe to apply in both argument and result positions.

After the first phase the system of constraints only contains inequalities either of the form $\pi \geq P$ or $\pi \geq u$. We simplify this further by replacing $\pi_1 \cap \pi_2 \geq \pi'$ with $\pi_1 \geq \pi' \wedge \pi_2 \geq \pi'$. This yields a system where each π is either a variable u, a subset of parties P, or a tuple $(\beta_1, \ldots, \beta_k)$. Next, we transform the system by replacing each $\pi \geq \pi'$ with a conjunction of projections $\bigwedge_{p \in \top}(\pi[\![p]\!] \geq \pi'[\![p]\!])$ where $\top$ indicates the set of all parties. We recursively collapse $(\beta_1, \ldots, \beta_k)[\![p]\!]$ to β_p, $(\beta_1 \wedge \beta_2)[\![p]\!]$ to $\beta_1[\![p]\!] \wedge \beta_2[\![p]\!]$, and $P[\![p]\!]$ to 1 if $p \in P$ or to 0 otherwise.

These transformations result in a system that consists of inequalities only of the form $\beta \geq \gamma$ where γ is either $u[\![p]\!]$ or 0 or 1. Now we replace inequalities of form $\beta_1 \wedge \beta_2 \geq \gamma$ with $\beta_1 \geq \gamma \wedge \beta_2 \geq \gamma$ to yield a system that only consists of inequalities of the form $\gamma \geq \gamma'$.

Let $\gamma \geq^\star \gamma' \iff \exists \gamma_1, \ldots, \gamma_n.\ \gamma \geq \gamma_1 \geq \ldots \geq \gamma_n \geq \gamma'$ be the transitive closure of our transformed system of inequalities. If $0 \geq^\star 1$ then the system is not solvable and we know that the original one was not solvable either. This corresponds to the case where we expect a value to exist for some party but it does not.

Let η be the type of the function that we have inferred, with the substitutions performed during constraint resolution already applied to it. Let $\mathrm{covars}(\eta)$ denote the free variables of η that can be found in covariant position and let $\mathrm{contravars}(\eta)$ denote ones in contravariant position. For every $u \in \mathrm{contravars}(\eta)$ let $P_u = \{p \mid u[\![p]\!] \geq^\star 1\}$. The constraint Φ of η is defined as the conjunction of all $u \geq P_u$ where u are contravariant.

The variables in covariant position may also be restricted. However, we cannot express these restrictions via constraints Φ as they are not rich enough. We need to consider two cases: first, if there exists a restriction $0 \geq^\star v[\![p]\!]$ then we know that v is not defined for party p in covariant positions; and second, if for some contravariant variable u we have $u[\![p']\!] \geq^\star v[\![p]\!]$ then v is defined for party p only if u is defined for party p'. These two rules give us a substitution scheme. We replace each v with $(\beta_1, \ldots, \beta_n)$ where $\beta_p = 0$ if $0 \geq^\star v[\![p]\!]$ and otherwise

$$\beta_p = \bigwedge \{u[\![p']\!] \mid u[\![p']\!] \geq^\star v[\![p]\!], u \in \mathrm{contravars}(\eta), p' \in \top\} \ .$$

The generated type is usually very difficult to read and should be simplified for displaying. For example, it is possible to remove duplicated conjunctions. However, note that it is not correct to replace a tuple $(u[\![1]\!], u[\![2]\!], u[\![3]\!])$ with u or otherwise from-expressions could be used on functional values.

5.3.4 Dynamic semantics of the protocol DSL

This section describes the dynamic behavior of protocol DSL programs. We do not specify how programs behave on concrete hardware or how network messages are ordered and scheduled. We are only looking to specify the ideal functionality protocol DSL programs.

The semantics of the language is relatively straightforward. We distinguish two kinds of values in the protocol DSL: functional values and tuples of primitive values where the i-th component denotes the value that the i-th party has. When we say that some value is undefined we mean that it is a tuple consisting of bottom values $\bot$. Because DSL programs have to terminate it is not possible to produce an undefined functional value.

The semantics is presented in small-step style. Transition rules are either from one expression to another ($\gamma \vdash e \xrightarrow{p} e'$) or from an expression to a value ($\gamma \vdash e \xrightarrow{p} v$). All transitions are annotated with probabilities (omitted if equal to 1). The meaning of constraints $[\![C]\!] \in \{\mathtt{true}, \mathtt{false}\}$ is defined in the obvious manner (C does not contain free type variables). The environment γ only contains the definitions of global functions and does not map local variables to values.

The expression evaluation rules are given in Figure 5.8. Mostly they are straightforward lambda calculus rules. We evaluate expressions under evaluation context $\mathcal{C}$ and substitute variables in case of function applications and let-expressions. Evaluation is performed strictly except for lambda (or type) abstractions and if-expressions. The from-expression rearranges the components of the tuple. For case-expressions all subexpressions are evaluated and then the correct components are picked out of the branches. The built-in function `rngBit` chooses each possible combination of bits with equal probability.

$$\frac{\dfrac{\gamma \vdash e \xrightarrow{p} e'}{\gamma \vdash \mathcal{C}[e] \xrightarrow{p} \mathcal{C}[e']}}{\gamma \vdash (\lambda x.\ e)v \to e[x \mapsto v]} \qquad \frac{\dfrac{x \in \mathrm{Dom}(\gamma)}{\gamma \vdash x \to \gamma(x)}}{\gamma \vdash (\Lambda\alpha.\ e)\tau \to e[\alpha \mapsto \tau]}$$

$$\frac{}{\gamma \vdash \texttt{let}\ x = v\ \texttt{in}\ e \to e[x \mapsto v]}$$

$$\frac{[\![C]\!] = \texttt{true}}{\gamma \vdash \texttt{if}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to e_1} \qquad \frac{[\![C]\!] = \texttt{false}}{\gamma \vdash \texttt{if}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to e_2}$$

$$\frac{}{\gamma \vdash \langle v_1, \ldots, v_n \rangle\ \texttt{from}\ \langle p_1, \ldots, p_n \rangle \to \langle v_{p_1}, \ldots, v_{p_n} \rangle}$$

$$\frac{\forall i \in \{1, \ldots, k\}.\ u_{p_i} = (v_i)_{p_i}}{\gamma \vdash (\texttt{case:}\ p_1 \to v_1, \ldots, p_k \to v_k) \to \langle u_1, \ldots, u_n \rangle}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ b_i \in \{0, 1\}}{\gamma \vdash \texttt{rngBit()} \xrightarrow{2^{-n}} \langle b_1, \ldots, b_n \rangle}$$

Figure 5.8: Semantics of the core language expressions

These semantic rules are the basis for our compiler but implementing them directly would result in an extremely inefficient evaluator. It would constantly compute values that are never used due to the case-construct dropping them, and often propagate bottom values, due to the case-construct introducing them. However, this is not a problem because we are compiling to an IR in the form of a finite directed acyclic graph with no control flow constructs. This allows us to eliminate all such inefficiencies with dead code elimination, by throwing away bottom values and the operations that have introduced them.

A program M consists of a sequence of top-level declarations $x_i : \sigma_i = e_i$ followed by a body of the program e. The environment γ is formed by mapping each x_i to e_i. We assume that e is an n-ary function from integers to integer. For input $x = (x_1, \ldots, x_n) \in \mathbb{N}_0^n$ the meaning of M is given by

$$[\![M]\!]_x(y) = \sum\{p_1 p_2 \ldots p_n \mid \gamma \vdash e\ x \xrightarrow{p_1, p_2, \ldots, p_n} y\} \ .$$

That is, for input x we look at the probability of reaching result y. The ideal functionality of program M is given by $[\![M]\!]$.

5.4 Low-level intermediate representation of the protocol DSL

5.4.1 Arithmetic circuits

Arithmetic circuits are the low-level IR for our protocol compiler. This representation is used for optimizations and compilation. An arithmetic circuit is a directed acyclic graph (DAG) where the vertices are labeled with operations and the incoming edges of each vertex are ordered. The input nodes of the circuit correspond to the representations of inputs to the arithmetic black box operation. In the case of protocol sets based on secret sharing, each input is represented by several nodes, one for each computing party. Similarly, the output nodes correspond to the shares of the output.

Communication between parties is expressed implicitly: each node of the circuit is annotated with the executing party, and an edge between nodes belonging to different parties denotes communication. This representation makes both computation and communication easily accessible to analyses and optimizations. It is straightforward to tell how many bits are sent in how many rounds and to find dependencies between computations. The fact that our DAG representation contains no control flow constructs or loops makes accurate analysis and powerful optimizations possible even on large graphs.

As an example, Figure 5.9 contains the graphical representation of the unoptimized DAG for the three-party additive multiplication protocol. Nodes denote operations and edges denote data dependencies. Inputs are denoted by x and y, the dollar sign denotes random generation, and nodes labeled with arithmetic operations denote the corresponding operations. The shape and color of a node indicates which party performs the operation: the first party is denoted with red circles, the second party with green squares, and the third party with blue diamonds. Edges running between two different shapes indicate network communication and are drawn as solid arrows. Output nodes have a double border. The DAG representation is typed but in this example we have omitted the types as they are all equal.

To compile a protocol specified in our DSL to an arithmetic circuit, all function calls have to be inlined and the code has to be converted to monomorphic form. If the control flow of a protocol requires some information about data known only at runtime, such as public inputs or lengths of dynamic arrays, then this protocol cannot be fully specified in the protocol DSL. In those cases SecreC can be used to specify parts of the protocol. Alternatively, protocols can still be written in C++, either fully or with some parts implemented in the DSL.

Semantics Because we do not formalize the syntax of our arithmetic circuits we also do not formalize their semantics. We assume that each m-input and n-

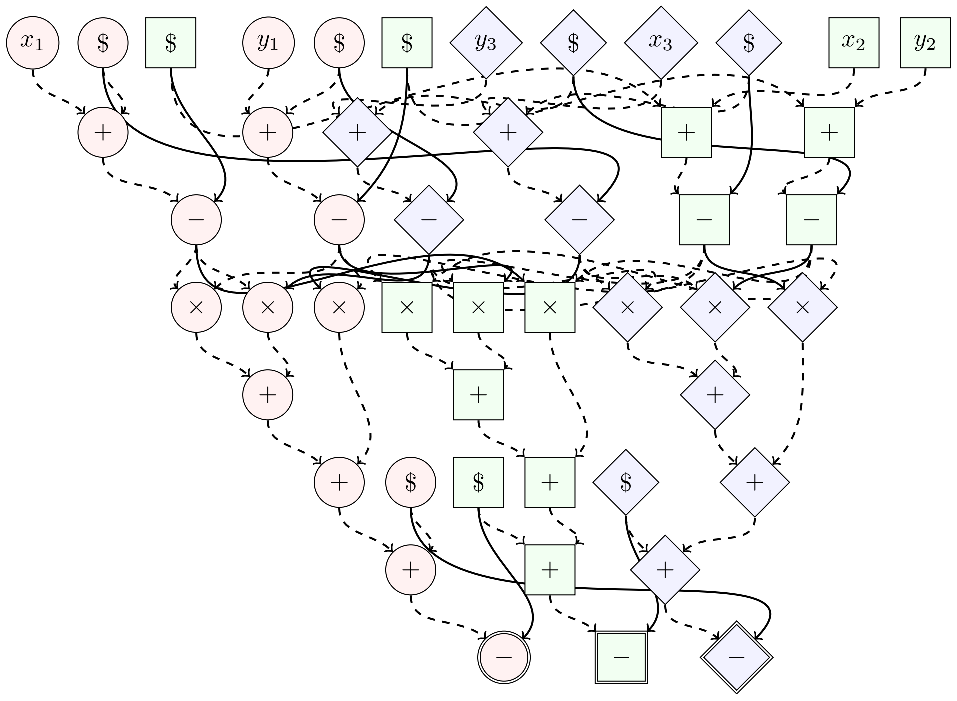


Figure 5.9: Unoptimized multiplication protocol DAG

output arithmetic DAG d has implicitly given meaning in terms of a function $[\![d]\!] : \mathbb{N}_0^m \to (\mathbb{N}_0^n \to [0,1])$. The semantics $[\![d]\!]$ does not encode any information about network communication between participants. It only describes the ideal functionality of the DAG d. For example, a DAG that performs 64-bit additive resharing between three parties has the following meaning for input (x_1, x_2, x_3):

$$f(y_1, y_2, y_3) = \begin{cases} 2^{-128}, & \text{if } \sum_{i=1}^{3} y_i = \sum_{i=1}^{3} x_i \mod 2^{64} \\ 0, & \text{otherwise.} \end{cases}$$

5.4.2 Optimizations

The IR is used to optimize the protocols. Due to the compositional nature of specification, the protocols typically contain inefficiencies such as constants that can be folded, duplicate computations, and dead code. So far, we have implemented all basic optimizations analogous to the ones reported in [70] for Boolean circuits: constant propagation, merging of identical nodes, and dead code removal. But as our biggest circuits only have a few hundred thousand nodes, and the arithmetic operations allow much more information about the computation to be easily gleaned, we have also successfully run more complex optimizations. We can simplify certain arithmetic expressions, such as linear combinations, even if communication is involved between operations.

In addition to generic optimizations from compiler implementation literature, we have implemented two optimizations that are not applicable to locally executed public code. First, we can eliminate network communication by sharing random number generators. Second, we can occasionally eliminate random numbers if doing so does not affect security.

Shared random number generators

Usually every input of a protocol is explicitly reshared to ensure that a party's view could be generated from only its inputs. Recall that the resharing protocol is implemented as follows: each party $\mathcal{P}_i$ generates a random value $r_i \leftarrow R$ and sends it to the next computing party $\mathcal{P}_{\mathsf{n}(i)}$, adds the generated value r_i to the input share $[\![u]\!]_i$, and subtracts the random number $r_{\mathsf{p}(i)}$ received from the previous computing party $\mathcal{P}_{\mathsf{p}(i)}$. The shares of the output $[\![v]\!]$ of the protocol are $([\![u]\!]_1+r_1-r_3, [\![u]\!]_2+r_2-r_1, [\![u]\!]_3+r_3-r_2)$. We see that $v = [\![v]\!]_1+[\![v]\!]_2+[\![v]\!]_3 = [\![u]\!]_1 + [\![u]\!]_2 + [\![u]\!]_3 = u$.

We can spot a pattern that occurs in resharing: a party generates a random number and sends it to some other party. This can be optimized by letting both parties generate the same random number using a common random number generator (RNG). The seed of the RNG needs to be agreed on beforehand. This optimization is not new and has previously been used in [63] and [78]. Our contribution is that our protocol DSL toolchain allows this optimization to be automatically applied, with no changes to the specification of protocols. The optimization itself is straightforward on our intermediate representation.

An analysis of generated intermediate code tells us that this technique reduces the network communication of three-party additive and XOR protocols by 45% to 50%, i.e. it nearly halves the required network traffic. For example, communication cost of integer multiplication is reduced by exactly 50%, and communication cost of 64-bit public division by 48%. Based on the network communication analysis and benchmarking from Section 5.6.5 we can tell that this translates into a performance improvement on large inputs. On scalar values, network latency is the performance bottleneck; thus, we do not expect a performance gain on small inputs.

We have also manually implemented this optimization for the multiplication protocol and compared the performance to that of the unoptimized manually implemented version to validate the effectiveness of this modification. We did not use the protocol DSL for this comparison because it was difficult to concurrently support two evaluation strategies.

We chose the multiplication protocol because of its simplicity, efficiency,

Table 5.1: Speedups of the shared RNG multiplication protocol over regular multiplication. The speedups have been measured for input vectors of sizes 1 to 10^8.

ℓ	Speedup on given input length								
	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
8	1.35	1.39	1.45	1.52	1.42	1.41	1.09	0.73	0.94
16	1.43	1.46	1.51	1.58	1.71	1.86	1.22	1.02	1.24
32	1.39	1.39	1.42	1.60	1.82	1.84	1.33	1.23	1.27
64	1.49	1.74	1.52	1.83	2.09	1.68	1.02	1.40	1.48

ubiquity in application, and because it is one of the least computation-heavy protocols. The comparison was performed using the methodology described in Section 5.6.1 and the results are displayed in Table 5.1.

We can see that performance has improved by up to two times. The optimization has not, in most cases, improved performance as much as was suggested by the reduction in communication. On small inputs the speedup is lower, as expected, due to the influence of network latency. Unfortunately, we do not have a good explanation why we do not see greater speedup on large inputs. A smaller speedup past 10^5-element inputs can be at least partially explained by the fact that we process only 10^5 elements at a time; thus, the number of rounds increases past that point. We see some significant slowdown that we are not able to explain for large input sizes in the 8-bit case.

Elimination of random numbers

When developing large protocols, it is occasionally very convenient to use existing private operations (partially) on constant values. However, using private operations directly on constant values introduces an inefficiency: classical optimization methods are not able to propagate constants through resharing calls. This problem can be solved manually by implementing optimized alternatives with public arguments but this is significant work. This problem has motivated us to implement an optimization to eliminate randomness nodes that are not needed for security.

It turns out that by using the security analysis algorithm from Section 5.5.2, we can detect if any randomness can be eliminated without affecting security. The detected nodes are then replaced with constant zero nodes. This optimization does not directly reduce network communication on its own, but facilitates other optimizations, like constant propagation and common subexpression elimination. Hence, the random number elimination step is performed before the rest of the optimizations. We have experimentally determined that applying this optimization iteratively with other optimizations has little to no effect.

Unfortunately, this optimization is not able to distinguish nodes that are needed to implement non-deterministic functionality from those intended for security. For implementing non-deterministic operations this optimization must currently be skipped. A more flexible solution would be to allow randomness nodes to be annotated if they are intended for security or not. So far we have not had the need to implement this functionality.

We have measured the effect of this optimization on the communication cost of various operations. For the first measurement we ran all optimizations except RNG elimination, while for the second measurement we first ran RNG elimination and then the rest of the optimizations. In all cases, except for very simple protocols like integer multiplication, the optimization reduced network communication. For all single-precision floating-point operations communication was reduced by 5% to 10%, and in the case of comparison by 13%. Improvement is smaller for 32-bit integer operations where we saw 3% to 6% reduction, with the exception of comparison that improved by 13.8%.

5.5 Security

5.5.1 Security definitions

To recall Section 2.3.1 let π denote a real functionality expressed as a collection of actors and $\mathcal{F}$ an ideal functionality expressed as a single actor. If π_{op} is an SMC protocol for performing a particular operation op on shared values, then the actors of π_{op} receive the shares of the inputs over the interface with $\mathcal{Z}$ at the beginning of the protocol. The adversary may corrupt some actors at the beginning of the protocol. During execution the actors exchange messages and compute the shares of the outputs. Corrupt actors send everything they receive to the adversary. Finally, the actors of π_{op} hand the shares of the outputs back to $\mathcal{Z}$.

A corresponding ideal functionality $\mathcal{F}^{\mathsf{op}}_{\mathtt{sec}}$ also receives the shares of the inputs from $\mathcal{Z}$ and corruption requests from $\mathcal{A}$. The functionality $\mathcal{F}^{\mathsf{op}}_{\mathtt{sec}}$ reconstructs the actual inputs from the shares, applies op to them and shares the results, thereby obtaining the output shares which it gives back to $\mathcal{Z}$. The functionality $\mathcal{F}^{\mathsf{op}}_{\mathtt{sec}}$ also sends the input and output shares of corrupt parties to $\mathcal{A}$. If the adversary is malicious, then it can also change the output shares returned to the corrupt parties.

An SMC protocol π_{op} is said to be secure (Section 2.3.2) if it UC securely implements the ideal functionality $\mathcal{F}^{\mathsf{op}}_{\mathtt{sec}}$. For example, the resharing protocol is a secure protocol for the identity function. Privacy is defined similarly—π_{op} is private if it UC securely implements $\mathcal{F}^{\mathsf{op}}_{\mathtt{priv}}$ where $\mathcal{F}^{\mathsf{op}}_{\mathtt{priv}}$ is obtained from $\mathcal{F}^{\mathsf{op}}_{\mathtt{sec}}$ by not sending the final shares to appropriate parties.

We know that sequential composition of two protocols, where the output shares from the first protocol are directly given as inputs to the second one without going through $\mathcal{Z}$, preserves privacy [18]. Against honest-but-curious adversaries, the composition of a private protocol and a secure protocol (commonly resharing) is secure [18]. Against malicious adversaries, privacy limits the amount of information they can learn about the inputs of the protocol [97].

A protocol optimization $\mathcal{T}$ transforms a protocol π implementing an ideal functionality $\mathcal{F}$ into a new protocol $\mathcal{T}(\pi)$ also implementing $\mathcal{F}$. Each optimization we have implemented in our DSL compiler *preserves the preservation of privacy*—we show that there exists an actor T such that

$$\forall \mathsf{Sim} : \big[\forall \mathcal{Z}, \mathcal{A} : \mathcal{Z} \| \pi \| \mathcal{A} \approx \mathcal{Z} \| \mathcal{F} \| \mathsf{Sim}^{\mathcal{A}} \Rightarrow \\ \forall \mathcal{Z}, \mathcal{A} : \mathcal{Z} \| \mathcal{T}(\pi) \| \mathcal{A} \approx \mathcal{Z} \| \mathcal{F} \| (\mathsf{T} \| \mathsf{Sim}_\alpha)^{\mathcal{A}}\big],$$

where Sim_α denotes Sim with renamed channels, such that it directly communicates only with T. In effect, if $\mathcal{T}$ is a protocol transformation, then $\mathsf{T} \| (\cdot)_\alpha$ is the corresponding *simulator transformation* used to transform a security proof of π into a security proof of $\mathcal{T}(\pi)$.

Unfortunately, the optimization for eliminating random numbers by replacing their sources with constant zeros does not fit into this framework. The randomness removal will not preserve the ideal functionality of transformed protocols. This is because the algorithm is not able to distinguish between randomness nodes that are redundant from the ones that are intended to offer some functionality.

5.5.2 Proving the security of protocols

Our protocol DSL contains no mechanisms to statically ensure the security or privacy of protocols. The type system of the language only verifies the lengths of the values but not their dependence on inputs or random variables. Hence, we also cannot speak about security-preserving compilation in the sense of [43].

We ensure security using data flow analysis on the low-level intermediate language. Our compiler pipeline contains the static analyzer by Pettai and Laud [97] that checks protocols expressed as arithmetic circuits for privacy against malicious adversaries. If a protocol passes that check and we know that it is followed by a resharing protocol, then it is also secure against honest-but-curious adversaries. The check is invoked after the translation from the protocol DSL to the intermediate language and the optimization of the generated intermediate code. Having the security check late in the pipeline ensures that the earlier optimizations do not introduce uncaught vulnerabilities.

The idea of the algorithm by Pettai and Laud [97] is relatively straightforward. It operates on arithmetic DAG described in Section 5.4. In the three-party case one of the participants is designated as an adversary $\mathcal{A}$. First, the DAG is transformed to what they call an *active-adversarial DAG* (AADAG). This is done by designating all inputs from $\mathcal{A}$ as special adversarial source nodes having no inputs and exactly one outgoing edge, and all values sent to $\mathcal{A}$ go into a special adversarial sink node that has no outgoing edges. It is clear that if all the data that is sent to $\mathcal{A}$ is directly randomly generated then this protocol is secure against an active $\mathcal{A}$. The key idea behind the algorithm is to transform the AADAG so that all values sent to $\mathcal{A}$ become random but the values observed by $\mathcal{A}$ do not change. If the AADAG can be modified in such a way then the protocol is secure against an active $\mathcal{A}$. The algorithm is presented in full detail in [97, Figure 3].

Our experience with the protocol DSL validates the security aspects of the language and compiler design. Indeed, we have found that implementing protocols in a secure manner is very straightforward. This can be explained by the fact that most protocols are compositions of simpler ones. When writing in this style, the UC theorem automatically provides the privacy guarantee. It is very rare that a protocol is added that is not purely a composition, and even in this case the automatic privacy checker is there to provide a safety net and validation for the programmer. In fact, for those reasons, to implement efficient and secure protocols in the protocol DSL one does not need to have a deep understanding of the security framework of the additive secret sharing scheme. Due to universal composability and compositional nature of protocols a developer almost never has to prove that a protocol is secure. But regardless, privacy of resulting protocol is always automatically verified.

Finally, we can claim that the protocol language can and has been used to build secure arithmetic black box.

Theorem 5. *Let M be a DSL program and d an arithmetic DAG such that $[\![M]\!] = [\![d]\!]$. If algorithm by Pettai and Laud [97, Figures 2 and 3] deems d private then there exists a protocol π that is a private implementation of $[\![M]\!]$ against active adversaries.*

Proof. We know that if DAG d is deemed private by the algorithm then there exists protocol π corresponding to d that is private against active adversaries [97, Theorem 1]. In another words π is private implementation of $[\![d]\!]$. Thus, π is also a private implementation of $[\![M]\!]$ against active adversaries. $\square$

Theorem 6. *The protocol DSL can be used to build arithmetic black box that is secure against passive adversaries. We have done so for three-party additive secret sharing.*

Proof. In Section 5.1 we saw how to implement addition, multiplication and conversion operations. These are sufficient for forming an ABB. The previous theorem gives that if functionality is preserved by compiling DSL program M into a DAG d and d has been shown to be secure then we can construct a protocol that implements M privately. We also know that private implementations can be transformed [18] into secure implementations against passive adversaries. $\square$

5.6 Experimental results

In this section we explore the performance of additive three-party protocols implemented in the DSL. For some protocols we also compare the results against our previous generation of protocols implemented in C++. In most cases we are not comparing algorithmically identical protocols. This is unavoidably so, as a key aim of the DSL is to simplify the development, optimization and exploration of protocols. The protocol implementations in C++ are long (for example, the C++ division protocol spans over 1500 lines of code) and difficult to read. It is often not clear if the C++ implementation matches the specification, and frequently it does not as the concrete implementations tend to employ many undocumented optimizations.

Another factor that makes identical comparison difficult is that the DSL floating-point protocols provide better numeric accuracy guarantees and operate correctly on a larger range of inputs. For instance, we found that some of the C++ protocols do not handle 0 properly, some operations have poor relative errors, and double-precision floating-point numbers provide very poor accuracy in the range of 10^{-7} when the order of 10^{-15} is expected. These differences should give a performance advantage to the C++ protocols. Providing a perfectly fair comparison would mean either incorporating defects into the DSL protocols or improving the C++ protocols. In both cases valuable time would be wasted.

As it currently stands, all floating-point protocols—addition, multiplication, inverse, square root, exponentiation, logarithm, sine, error function, and conversion and comparison operations—take less than 900 lines of code combined in the DSL. The C++ protocol suite offering less functionality has over 5000 lines for floating-point operations with additional 2000 lines of support code for various fixed-point helper protocols. Notably, the protocol suite for logarithmic numbers (addition, multiplication, inverse, square root, exponentiation, logarithm) developed in [41] takes less than 300 lines of DSL code. This is thanks to the ability to re-use generic operations such as fixed-point polynomial evaluation developed for floating-point protocols.

5.6.1 Benchmarking methodology

Benchmarking was performed on a dedicated cluster of three computers connected with 10Gbps Ethernet. Each computer was equipped with 128GB DDR4 memory and two 8-core Intel Xeon (E5-2640 v3) processors, and was running Debian 8.2 Jessie. Both memory overcommit and swap were disabled. During benchmarking only the necessary system processes and some low-overhead services (such as SSH and monitoring) were enabled. In every case only a single computation thread was used other than random number generation that was offloaded into separate threads.

Even on large input sizes, and even despite the fact that our hardware and platform supports it, we have not parallelized any protocol computation to multiple cores or used the networking layer in a parallel manner. We have limited ourselves to benchmarking on a single compute core to reduce the number of tunable parameters and keep the performance presentation simple.

Every protocol was benchmarked on various input sizes. When executing a protocol on many inputs, the round count remains the same while the amount of network communication increases. On a decent network connection, the evaluation of the multiplication protocol on scalars, and on vectors of length 10000, takes roughly the same time. On every input length we performed at least ten repetitions and, to reduce variance, significantly more on small input lengths (up to 5000 repetitions).

To estimate the execution time of a protocol on a specific input length we computed the mean of all measurements on that length. A single running time measurement was computed by taking the running times for all computing parties and finding the maximum of those. This is necessary as asymmetric protocols will terminate faster for some participants. The maximum reflects the time it takes for the result of the operation to become available to all participants.

We found that running tests ordered by ascending input length gave significantly better performance results than running them in randomized order. Sequential order results in a steady increase of network load which is predictable for the networking layer but is not a very realistic scenario for SMC applications. For this reason, for each operation we performed measurements in randomized order. When evaluating the performance of an SMC system, we find that performing tests performing tests in an order that is predictable for the networking layer is unacceptable and easily leads to dubious performance results.

To compare DSL protocols against previously implemented C++ protocols we computed the speedup on every input length by dividing the estimated execution time of the old C++ protocol with the estimated execution time of the respective DSL protocol. A speedup greater than one means that the new protocol was, on average, faster than the old one. All measurements were performed in an identical setup, using the same Sharemind version. Only the loadable module providing the protocols set was differed.

5.6.2 Speedup with respect to the previous generation of protocols

In order to understand the performance profile of the protocols, we first take a closer look at the performance of the floating-point multiplication protocol. In Figure 5.10 we can see the running time of the DSL and C++ protocols depending on input size. The x-axis denotes the input size and y-axis the running time. Notice

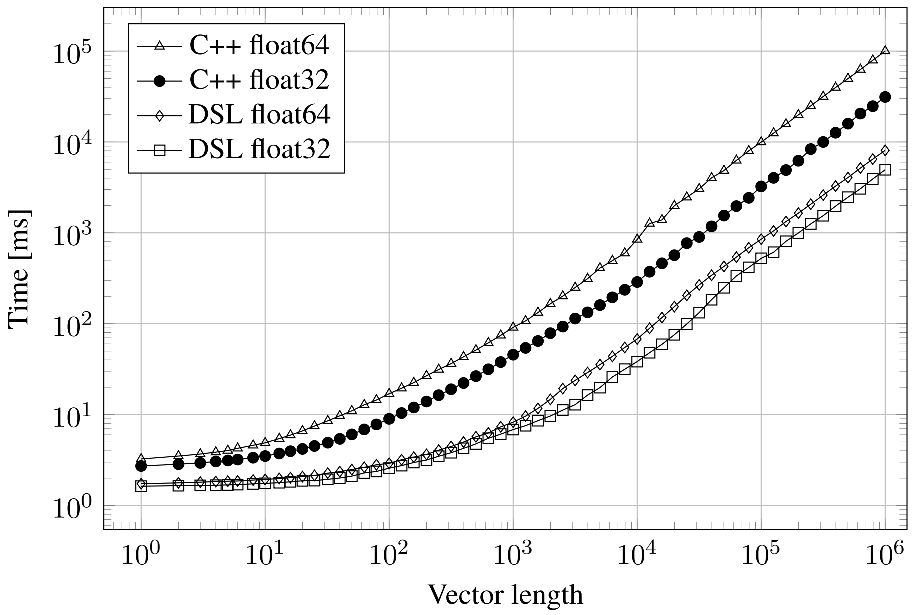


Figure 5.10: Running time of the floating-point multiplication protocol

how the running time is roughly constant up to around 500 elements after which it grows linearly. We call this point the *saturation point* because after this point execution time is no longer latency bound and is limited by some other factors (such as bandwidth or local computation). Unfortunately, this figure only tells us that the DSL version is faster than the C++ version but the graph does not clearly express by how much.

The performance difference is expressed more clearly in Figure 5.11 that shows the speedup of DSL protocol compared to C++ versions. We can see that the new floating-point multiplication is two to twelve times faster. It is interesting to note that the speedup is not constant and depends on input size. This is because performance on small inputs is dominated by network latency and performance on large inputs by network bandwidth. Hence, on small inputs the running time is mostly determined by the round count and on large inputs by the amount of data sent over the network. Looking at the speedup graph, it is reasonable to postulate that single-precision floating-point multiplication now has about two times less rounds and performs six times less network communication. Experimentally, we found single-precision floating-point multiplication to take about 9.5 times less communication. Unfortunately it is difficult to tell how many communication rounds protocols implemented in C++ perform.

For the rest of the operations we will not give such a detailed performance

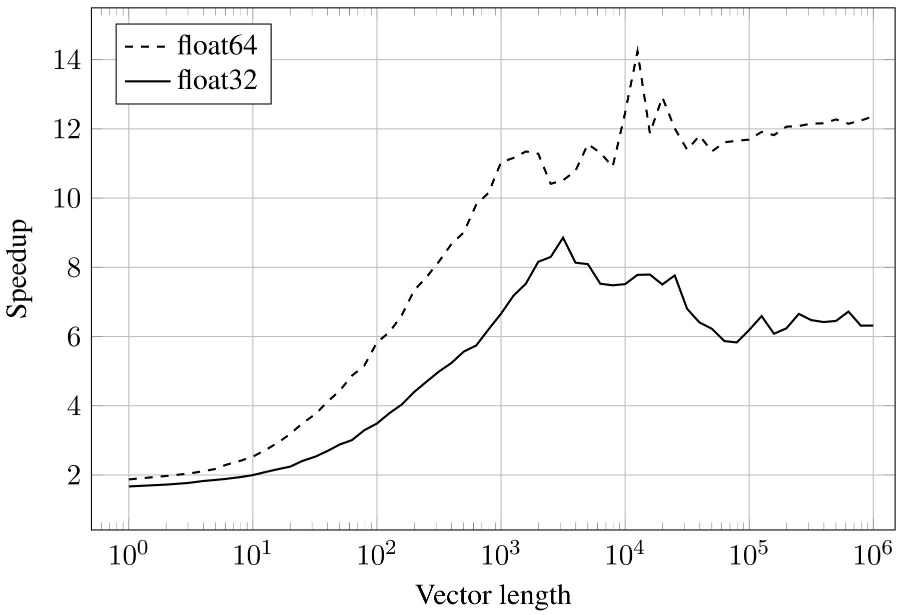


Figure 5.11: Speedup of floating-point multiplication implementation in the DSL with respect to the implementation in C++

analysis. Instead, we only present the measured speedup for various input sizes. The results of comparisons with protocols implemented in C++ are presented in Table 5.2. Integer operations have been benchmarked on 32-bit integers and floating-point protocols on single-precision numbers. We chose to display only 32-bit versions because some 64-bit integer operations are not implemented in C++ and for the rest of the operations, the results are quite similar, mostly favoring DSL protocols. We can see that shift-right by a private value ($a \gg b$) has benefited a lot from a redesign in the DSL. Similarly, the redesign of the floating-point error function has improved performance drastically.

We can see that for every floating-point operation the use of the protocol DSL has resulted in better performance. All new floating-point protocols run at least twice as fast and in many cases the DSL provides an over 10-fold speedup. Some slowdown for integer division operations was to be expected, as they are very well tuned medium-sized protocols. Surprisingly, past 10-element inputs we can see some speedup in favor of our compiler.

The performance gain of the multiplication protocol comes from the use of shared random number generators, as our previous generation of protocols did not take advantage of such optimization. In the speedup table we have presented a comparison with the DSL protocol and we see a speedup of up to 1.9. In the specific case of integer multiplication we actually continue to use the manually optimized

Table 5.2: Speedup in comparison with non-DSL protocols

Op.	Speedup on a given input length					
	10^0	10^1	10^2	10^3	10^4	10^5
$a \mathbin{/} b$	1.0	1.2	1.7	1.8	1.6	1.7
$a \times b$	1.4	1.4	1.4	1.7	1.9	1.6
$a \gg n$	1.1	1.1	1.6	3.3	4.4	3.0
$a \mathbin{/} n$	0.4	0.5	0.9	2.4	4.0	4.3
$a < b$	0.8	0.9	1.4	3.3	6.2	5.4
$a \gg b$	1.4	2.6	10.2	31.6	31.4	26.9
$x + y$	2.6	3.0	4.8	5.5	4.1	3.9
$x \times y$	1.7	2.0	3.5	6.7	7.5	6.2
$x < y$	6.9	7.4	8.5	8.5	6.8	6.6
$\sin x$	7.8	8.9	13.2	11.1	8.0	8.6
$1 \mathbin{/} x$	2.1	2.9	6.3	10.0	9.4	9.5
$\ln x$	13.0	14.8	20.0	15.2	11.5	11.7
$\sqrt{x}$	2.6	3.7	8.4	13.7	13.2	12.6
e^x	5.5	7.5	17.4	25.5	27.0	23.1
$\operatorname{erf} x$	5.0	13.9	44.9	74.6	91.8	88.5

Note: a and b denote 32-bit additively shared integers; n denotes a 32-bit public integer; x and y denote single-precision private floating-point numbers.

C++ protocol. By hand-tuning we can better optimize memory usage and tailor the protocol to use communication patterns that are more suitable for our network layer. We have measured that in the best case scenario the manually implemented multiplication protocol seems to have a speedup of 1.2 over the DSL generated one. This happens only in few cases. Mostly the protocols perform roughly equally well.

5.6.3 Performance of elementary operations

The performance of multiplication protocol is presented in Table 5.3. We have evaluated the performance of integer multiplication on up to 10^8-element input vectors. We are able to perform up to 17 million 64-bit multiplications per second and up to 27 million 32-bit multiplications. Note again that performance is measured for a single computation thread. With more parallel threads we could saturate the communication channels.

Table 5.4 shows the performance of other elementary integer operations. We have benchmarked less-than comparison, right-shift and division operations on up to 10^6-element input vectors. We also consider the case where the second

Table 5.3: Performance (in 1000 operations per second) of ℓ-bit integer multiplication.

ℓ	10^0	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
8	7.89	78.9	776	6860	31 400	52 900	48 800	33 300	45 800
16	8.23	82.2	794	6710	28 900	48 300	37 000	33 500	43 000
32	7.58	75.3	723	5790	20 800	27 400	23 700	24 600	26 200
64	7.34	72.1	677	5110	13 800	14 600	10 500	16 100	17 200

argument of right-shift or division is a public integer. Public divisions and shifts are quite common. The performance advantage over completely private operations is considerable. In the case of 64-bit integers we can perform up to a few hundred thousand operations per second. The slowest operations is division of two private integers which achieves ten thousand operations per second.

Performance results for floating-point operations are presented in Table 5.5. We have measured addition, multiplication, comparison, inverse, square root, exponentiation, natural logarithm, sine, and error function from single-element inputs to 10^6-element input vectors. The results have been presented in thousands of operations per second. Looking at the table, it is clear that performance scales very well with vectorization: only a few hundred scalar operations can be executed per second but by computing on many inputs we can perform hundreds of thousands of operations per second. Generally, floating-point operations are much slower than integer operations. For example, multiplication is over a hundred times slower. However, integer division is a very expensive operation and applications that need to do many of them should consider floating-point numbers. In fact, floating-point inverse is twice as fast as integer division of comparably sized types.

Performance results for fixed-point operations are presented in Table 5.6. Performance-wise they have two significant advantages over floating-point numbers. First, addition is local. Second, multiplication is over twice as fast. For this reason we suggest using fixed-point numbers when possible over floating-point numbers. Of course, fixed-point numbers are not usable in all applications. They are subject to overflows and underflows, and are not suitable if good relative error is required. Furthermore, operations other than addition and multiplication are much slower than the respective floating-point ones.

5.6.4 Performance comparison of applications

We have also benchmarked private satellite collision analysis from [62] using the new protocols. We see a roughly 5-fold speedup, going from 0.5 satellite pairs per second to 2.5 pairs per second. When processing 100 pairs in a vectorized manner, we gain a roughly 8-fold speedup, going from 0.7 pairs per second to 6 pairs

Table 5.4: Performance (in 1000 operations per second) of integer operations.

Op.	ℓ	1000 OP/s on given input length						
		10^0	10^1	10^2	10^3	10^4	10^5	10^6
$a \gg n$	8	1.31	13.6	133.2	1062	3903	5785	6473
	16	1.22	12.0	115.2	897	3146	3642	3932
	32	1.61	15.6	137.9	833	2005	1583	1821
	64	1.43	13.9	113.8	548	820	678	821
$a \gg b$	8	0.73	6.9	67.7	539	1717	2067	2173
	16	1.18	12.1	109.1	604	1287	1079	1312
	32	1.03	10.3	86.4	409	463	422	502
	64	0.63	6.0	48.7	111	122	142	153
$a < b$	8	1.06	10.4	95.8	584	1563	1925	2114
	16	1.25	12.2	93.3	490	1171	1081	1188
	32	0.80	7.7	60.8	344	748	554	624
	64	0.74	7.0	53.2	250	335	263	296
$a \mathbin{/} n$	8	0.82	8.1	73.2	462	1528	2082	2161
	16	0.57	5.5	48.9	307	1056	1108	1206
	32	0.51	5.0	44.5	255	630	531	590
	64	0.48	4.4	32.2	150	239	188	196
$a \mathbin{/} b$	8	0.75	7.0	53.7	191	260	195	205
	16	0.66	6.1	42.9	128	140	121	126
	32	0.52	4.0	13.6	21	20	25	26
	64	0.43	2.5	6.1	7	8	10	10

Note: ℓ denotes the bit width of the inputs, a and b denote additively shared integers and n denotes a public integers.

Table 5.5: Performance (in 1000 operations per second) of optimized floating-point operations. Variables x and y denote private floating-point numbers.

Op.	Prec.	1000 OP/s on given input length						
		10^0	10^1	10^2	10^3	10^4	10^5	10^6
$x < y$	single	1.38	12.98	87.9	217.4	230.9	206.8	225.4
	double	1.30	11.52	67.5	136.4	118.5	108.4	120.5
$x \times y$	single	0.61	5.71	38.8	145.9	261.1	190.9	202.0
	double	0.58	5.17	34.3	121.2	147.2	117.3	123.7
$x + y$	single	0.29	2.55	14.9	32.2	31.6	29.6	31.9
	double	0.25	2.07	9.2	13.2	13.5	14.5	15.8
$\sqrt{x}$	single	0.31	2.81	18.2	55.9	66.9	56.8	57.4
	double	0.24	1.96	7.4	12.7	12.4	13.8	14.3
x^{-1}	single	0.32	2.86	18.3	54.7	63.2	55.6	56.7
	double	0.24	2.06	9.3	18.1	19.4	19.9	20.7
$\exp(x)$	single	0.22	2.01	13.5	35.0	45.3	37.4	39.1
	double	0.16	1.38	6.0	11.4	11.8	12.3	12.5
$\mathrm{erf}\, x$	single	0.25	2.23	13.5	29.2	32.3	29.1	29.5
	double	0.19	1.43	4.7	7.0	7.0	8.0	8.2
$\sin x$	single	0.13	1.13	6.0	10.2	10.0	10.1	10.6
	double	0.12	0.93	2.8	3.3	3.3	4.0	4.3
$\ln x$	single	0.13	1.12	5.6	8.4	8.5	8.2	8.6
	double	0.13	0.91	2.7	3.2	3.1	3.5	3.8

Table 5.6: Performance of fixed-point operations in 1000 operations per second.

Op.	ℓ	1000 OP/s on given input length						
		10^0	10^1	10^2	10^3	10^4	10^5	10^6
$s \times t$	32	0.95	9.17	73.4	334.3	524.7	462.2	488.0
	64	0.92	8.39	60.9	228.4	266.9	237.1	243.6
$\sqrt{s}$	32	0.30	2.53	11.4	19.8	21.0	23.9	23.9
	64	0.22	1.57	4.3	5.6	5.9	7.5	7.8
s^{-1}	32	0.24	1.60	4.2	4.4	6.2	7.7	7.8
	64	0.17	0.80	1.0	1.3	2.0	2.3	2.4

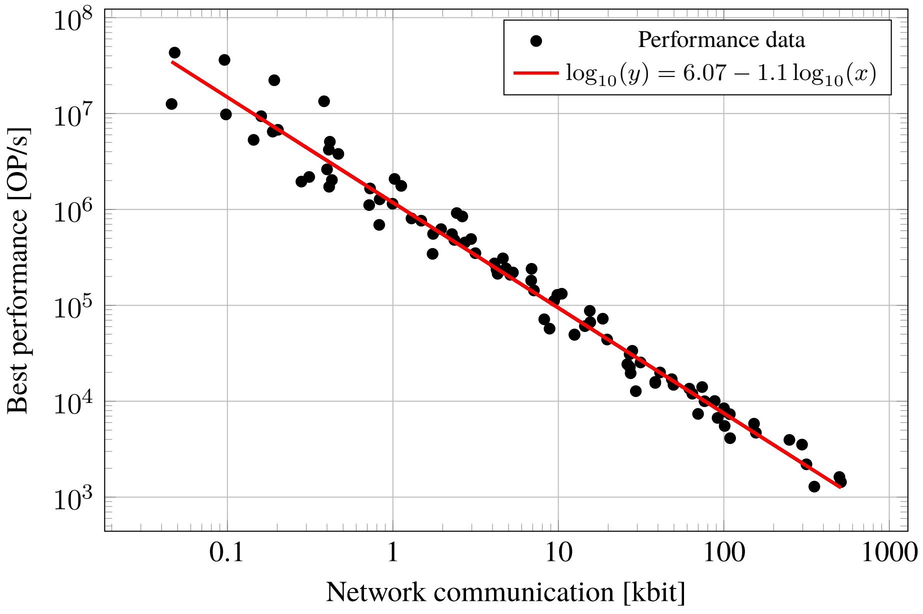


Figure 5.12: Best performance vs network communication

per second. This confirms the obvious—improving the performance of low-level operations has a great effect on the performance of high-level applications. We have measured the performance of the privacy-preserving social study from [17] on old and new protocol sets. We only saw about 20% performance improvement on a local cluster. Later investigation revealed that performance was bottlenecked by network latency due to an implementation error that caused many sequential Boolean conjunctions to be performed.

5.6.5 Performance estimation

The performance of DSL generated protocols can be relatively well estimated without ever running the actual protocols. In Figure 5.12 and Figure 5.13 we have graphed the benchmarking results of all our DSL protocols, including those for logarithmic numbers and golden section numbers from [41]. Figure 5.12 plots the best performance (vectorized operations) compared to the protocols' communication cost. Figure 5.13 plots the worst performance (scalar operations) compared to the protocols' round count. We can see that the best performance can be very well estimated ($R^2 = 0.975$) linearly from communication cost and the worst performance from the number of rounds ($R^2 = 0.94$).

Those estimates are suitable only for our specific test setting (the version of Sharemind, networking configuration, hardware specification, operating system).

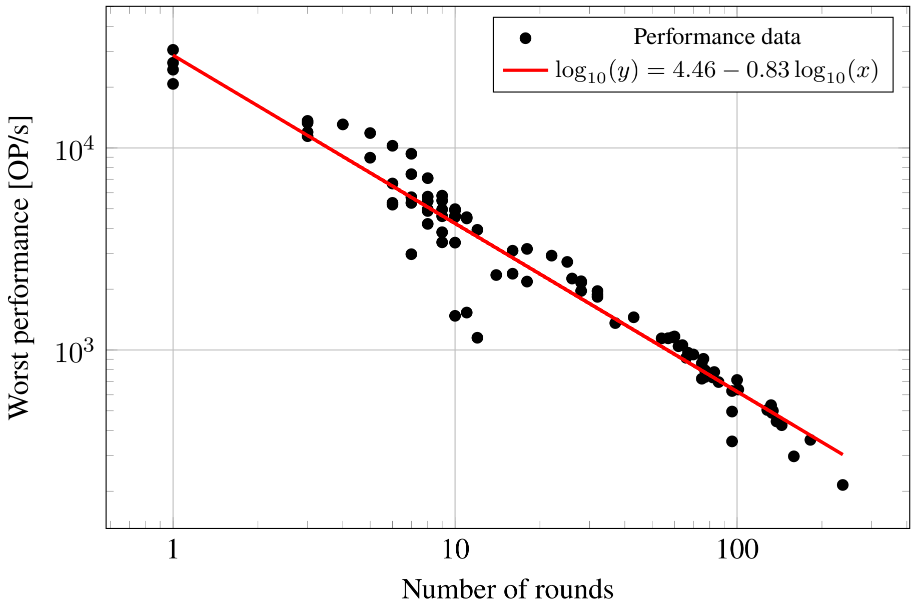


Figure 5.13: Worst performance vs number of rounds

We have not explored how to derive more general models or how to best estimate performance on arbitrary input lengths. We postulate that very accurate general performance models could be derived that also account for the concrete round structure and give estimates for any input length and network setting. This remains future work.

5.7 Discussion and future improvements

The protocol DSL has only been used to implement additive three-party protocol set. Implementing other security schemes with a fixed number of participants should not be too difficult. We have considered the implementation of a protocol suite based on Beaver's multiplication triples, or the information-theoretic three-party scheme that uses correlated randomness by Araki et al. [2]. In the second case, no modifications to protocol DSL language or supporting infrastructure are needed. In fact, we can implement the bitwise conjuction protocol as presented in Listing 5.17. Compared to the additive three-party multiplication in Section 5.1 the multiplication by Araki et al. requires two times less network communication but twice as much storage. Integer multiplication is also straightforward to implement in the DSL but requires the computation of 3^{-1} in $\mathbb{Z}_{2^n}$ which makes it less elegant in presentation.

The multiplication protocol using Beaver triples can be implemented very

Listing 5.17: Bitwise conjunction by Araki et al.

```
struct xorShare[n] = {
    x: uint[n]
    a: uint[n]
}

def xorCorrelated () : uint[n] = xorReshare(0)

def conj (u: xorShare[n], v: xorShare[n]): xorShare[n] = {
  let
    c = xorCorrelated()
    r = u.x & v.x ^ u.a & v.a ^ c
    z = r ^ (r from Prev);
  xorShare {z, r}
}
```

straightforwardly in the DSL but it would require minor modifications to the language and more significant modifications to Sharemind back-end and the way LLVM code is generated. Namely, we would need a process for multiplication triple pre-computation. For a particular protocol in DAG form it is easy to tell how many triples of which size are needed, but it is not as easy for high-level applications. Falling back on generating triples during online execution would incur a very high network communication cost compared to the additive three-party scheme.

Currently the protocol DSL does not support security schemes with a flexible number of computing parties. For example, it is not possible to specify generic Shamir's secret sharing scheme. This restricts the set of security schemes that we can implement quite significantly but it has not been a limitation so far because we have focused on the additive three-party protocol suite. However, this restriction could limit wider adoption. One solution is to adopt the concepts of share and wire types from Wysteria [100].

Unfortunately, the protocol DSL does not support data types of dynamic size. For this reason, it is impossible to implement, say, a protocol for aggregating arbitrary dynamic arrays. Similarly, we do not support dynamic control flow branching based on public values. One case where this has resulted in sacrificing performance is the public division protocol where we could send less network data for certain divisors. However, supporting dynamically sized types would mean that our intermediate representation could no longer supply precise information on network communication cost. For now we have found that the ability to combine the DSL-implemented pieces with either C++ or SecreC gives us sufficient flexibility.

Regardless of the limitations, the protocol DSL has served us well. We have demonstrated an order of magnitude performance improvement with respect to our previous protocol set and we have shown that developing secure protocols in the DSL is a much better experience than manual implementation in C++ or some other general-purpose language. This is not only due to the domain-specific design choices we have made but also thanks to the tools that we support.

CHAPTER 6

PRACTICAL APPLICATIONS AND PROTOTYPES

Most of the applications that have been developed on Sharemind use SecreC 2 to some degree. This chapter gives a short overview of several applications and describes how SecreC 2 was used in them. For each applications we explain the problem that was solved, give a general overview of the solution, and explain what role SecreC 2 played in the application. We can confidently claim that, as it stands, SecreC 2 has the most real-life applications and the largest code base out of all languages intended for SMC.

6.1 The private satellite collision analysis prototype

Kamm and Willemson [62] implemented secure floating-point arithmetic and developed an oblivious satellite collision analysis algorithm. Using the secure floating-point primitives, the algorithm calculates the probability of a collision between two satellites given their trajectories. The underlying secure operations (addition, multiplication, inverse, square root, natural exponentiation function, error function) were implemented in C++ but the algorithm itself was developed in SecreC 2. The floating-point protocols were later re-implemented using the protocol DSL. In fact problems with C++ implementation were a large motivation for creation of the protocol DSL.

Finding the probability of collision between two spherical objects moving in an orbit is a non-trivial task. To see that, we give a brief overview of one of the steps of the algorithm. After converting input data to a more convenient form the algorithm computes the encounter probability p as the following double integral:

$$\frac{1}{2\pi\sigma_x\sigma_y}\iint_T \exp\left(\frac{-1}{2}\left[\left(\frac{x-x_m}{\sigma_x}\right)^2+\left(\frac{y-y_m}{\sigma_y}\right)^2\right]\right) dydx \ ,$$

where R is the combined radius of the bodies, (x_m, y_m) is the center of the bodies in the encounter plane, σ_x and σ_y are the lengths of the semi-principal axes, and $T = [-R, R] \times [-\sqrt{R^2 - x^2}, \sqrt{R^2 - x^2}]$. The value of the integral is numerically approximated using Simpson's rule:

$$p \approx \frac{\Delta x}{3\sqrt{8\pi}\sigma_x} \left(f(0) + f(R) + \sum_{i=1}^{n} 4f((2i-1)\Delta x) + \sum_{i=1}^{n-1} 2f(2i\Delta x) \right) \quad ,$$

where $\Delta x = \frac{R}{2n}$ and the function f is defined as:

$$f(x) = \left[\operatorname{erf}\left(\frac{y_m + \sqrt{R^2 - x^2}}{\sqrt{2}\sigma_y} \right) + \operatorname{erf}\left(\frac{-y_m + \sqrt{R^2 - x^2}}{\sqrt{2}\sigma_y} \right) \right] \times \left[\exp\left(\frac{-(x + x_m)^2}{2\sigma_x^2} \right) + \exp\left(\frac{-(-x + x_m)^2}{2\sigma_x^2} \right) \right] \quad .$$

The oblivious SecreC implementation only spans around 500 lines of code not including the reusable components from the standard library. The algorithm admits to vectorization very easily. For example, when computing the final probability approximation p, all of the calls to function f can be evaluated in parallel. The native support for SIMD style operations greatly simplifies the implementation. The C++ code for secure operations spanned about 4500 lines.

Kamm and Willemson implemented two versions of the algorithm. The first version only processes one satellite pair at a time. The second version was parallelized to process n satellite pairs at a time. Unfortunately, the reported speedup for the vectorized case was rather small (about three times). This can be attributed to the algorithm already operating on quite large vectors even when processing a single satellite pair.

Performance was greatly improved by using the floating-point operations implemented with the protocol DSL: about fivefold in the scalar case and eightfold in the vectorized case. The performance improvements are from the DSL generated protocols alone and not from any improvements to the networking layer. When combined with the improved networking layer, we see the amortized time for processing a single pair going from a minute to a few hundred microseconds.

During development Kamm and Willemson implemented various reusable components. The vector and matrix operations can be used in other algorithms and across different protection domain kinds that support basic floating-point operations. Kamm and Willemson implemented vector operations such as length, dot product, and cross product; product of two-dimensional matrices; some special operations for diagonal matrices and covariance matrices; and eigensystem computation for 2×2 matrices. For many of these operations Kamm and Willemson also implemented SIMD parallel versions. Most of the operations can be found in the SecreC 2

Listing 6.1: Parallelised dot product for floating-point vectors in SecreC 2

```
template <domain D>
D float64[[1]] dotProduct (D float64[[2]] x, D float64[[2]] y) {
    assert (shapesAreEqual (x, y));
    return rowSums (x * y);
}
```

standard library in the `matrix` module. The development of the standard library has since continued. For example, the current implementation of the vectorized dot product (Listing 6.1) differs greatly from the one presented by Kamm and Willemson [62, Fig. 9].

6.2 The privacy-preserving social study

In Estonia, the information and communication technology sector is a rapidly growing industry where skilled and educated specialists are in demand. Wages in the industry continue to raise and academia cannot match the salaries. This causes friction. For instance, the universities in Estonia have formed a hypothesis that students who work during their studies do not graduate in nominal time and many of them quit before graduation.

This hypothesis can be verified through a study that links tax and education records. However, conducting the study with plain data would normally be impossible, as Estonian data protection and tax secrecy legislation does not allow linking these records. Bogdanov et al. [17] used Sharemind to perform this study securely. The study was conducted by linking the database of individual tax payments from the Estonian Tax and Customs Board and the database of higher education events from the Ministry of Education and Research. The analysis processed ten million secret shared tax records and half a million secret shared education records. This was the largest cryptographically private statistical study ever conducted on real data. A more detailed overview (including regulatory, organizational and social aspects) of the study can be found in [108].

The study was implemented using the Rmind [9] statistical analysis system. It was designed to mimic the statistical analysis system R. Rmind is a remote application running on client side. To executing commands the client sends requests to the Sharemind computing parties that execute the requests via a compiled SecreC 2 program. Rmind heavily relies on the SecreC 2 script in the sense that all private operations that Rmind can invoke have been implemented in SecreC 2. This includes primitive arithmetic operations, statistics operations, array manipulation, secure database manipulation and data aggregation. We have measured that, as it stands, Rmind uses about ten thousand lines of SecreC 2 code. This does not include the standard library that contains many of the statistics operations and also offers the database interface that Rmind heavily uses. The study was implemented using about few thousand lines of Rmind code.

Notably, the social study used sorting, database linking and database aggregation which were all implemented in SecreC 2. The sorting algorithm was by Bogdanov, Laur and Talviste [21] and database linking by Laur, Talviste and Willemson [79]. The database aggregation algorithm was presented in [17].

6.3 A privacy-preserving survey system

Surveys often collect sensitive data. For example, employee satisfaction surveys usually contain questions that employees might not want to answer personally to the employer. They would prefer to remain anonymous. These surveys usually require employees to trust either the employer or some third party to keep their answer secret. This kind of trust can be easily betrayed, accidentally or maliciously. Secure computing can be used to make survey systems more trustworthy by collecting responses in encrypted (or secret shared) form and analyzing the encrypted values without revealing any actual answers of concrete individuals. Only aggregate results are revealed.

During the FP7-funded PRACTICE project, Partisia, Cybernetica AS, and the Alexandra Institute implemented a survey system prototype to facilitate the collection of sensitive data to produce useful statistics without allowing third parties to see the individual answers. The overview of the system's design is presented in [111]. Cybernetica AS, in partnership with the Alexandra Institute and Partisia, deployed the secure survey system for the Tartu City Government[1]. This is the first real-life application of the Sharemind secure survey system. The system has also been used to conduct internal employee satisfaction surveys in Cybernetica AS in 2014 and 2015 [108].

The survey system has a web-based user-facing front-end. However, the survey results are aggregated and analyzed using a SecreC 2 script. We measured that the application only uses about 500 lines of SecreC 2 code, heavily relying on the functionality provided by the standard library. Front-end was implemented in JavaScript (9000 lines) and back-end in Java (21 thousand lines).

[1]`https://practice-project.eu/blog/entry/pilot-of-the-secure-survey-system-created-in-practice` (April 2017)

6.4 The tax fraud detection prototype

The Estonian government loses significant revenue due to value-added tax avoidance. The Estonian Tax and Customs Board has estimated that 220 million euros are lost every year. To increase revenue the government proposed a change in legislation that made it compulsory for companies to report all significant financial transactions. The updated version of the proposal was later accepted, but the first version was initially rejected. One of the reasons for rejection was the concern for privacy as the database would contain many of the financial transactions in Estonia.

The privacy concerns can be alleviated using secure computing. The individual transactions can be kept secret but transactions of companies with fraud risk can be disclosed for further analysis. Cybernetica built a prototype application specifically for this purpose [15]. The application performs the risk analysis obliviously and in the end only reveals the companies with high fraud risk.

The data processing, aggregation and fraud risk analysis were implemented in SecreC 2. On a local cluster it took 43 minutes to process 2.6 million transactions from 2000 companies. In [15] it was estimated that a single month of transactions takes ten days to process, using about 20 thousand euros worth of hardware. Later in [16], the authors demonstrated that with algorithmic optimizations and map-reduce style parallelization, it is possible to process a month's worth of transactions (100 million transactions between 80 thousand companies) in just a few hours. Running the optimized setup on the cloud cost under 200 US dollars. These results demonstrate that deploying and running a large-scale application that performs secure computations on the cloud has become very cost-efficient.

We have measured that the tax fraud detection prototype running on the cloud uses just under three thousand lines of SecreC 2. However, most of the code is for database manipulation (e.g., creating tables, reading data, linking tables). The risk analysis and the associated aggregation algorithms take less than a thousand lines of code. The authors developed two versions of the risk analysis algorithm: a fast version that leaks some extra information, and a slower version that does not. The fast version relies on the assumption that a company cannot be identified from the number of its business partners. The ability to choose the acceptable level of leakage is an absolutely essential feature of SecreC 2 and, as demonstrated, can enable secure applications that would otherwise be too slow for for real-life scenarios.

6.5 The frequent itemset mining prototype

The frequent itemset problem is the task of finding sets of items that are frequently bought together. Such information can be used by automatic recommendation systems that offer to clients logical suggestions about other items they might be looking for. For example, a customer looking for eggs and flour might also be interested in milk. However, customer data is sensitive and customer may refuse to give away their personal information. More customer trust might be gained by performing *frequent itemset mining* (FIM) obliviously.

Bogdanov, Jagomägis and Laur [13] implemented standard frequent itemset mining algorithms in privacy-preserving manner. They implemented the Apriori [1] and Eclat [116] algorithms in SecreC 1 and measured their performance. They noted that Apriori performs better but Eclat uses less memory. The performance advantage of Apriori comes from the fact that it traverses the frequent itemset graph in breadth-first manner. Thus, each level can be processed in parallel. In [14] the same authors implemented a hybrid combination of Apriori and Eclat to improve memory usage while performing comparably to Apriori.

The algorithms in [13, 14] were implemented to only handle dense data, meaning that memory requirements increased linearly with the product of the number of purchases and the number of available items. In reality, each purchase includes only a small number of items. It is difficult to use this assumption in the oblivious setting because we want to hide the number of items in every basket.

Laud and Pankova [75] used SecreC 2 to implement frequent itemset mining for both sparse and dense data. The authors implemented various different FIM algorithms and presented their round and communication complexity analysis. All algorithms were implemented in both sparse and dense variants. The performance measurements [49, Section 2.3.7] indicate that the sparse variants generally perform less network communication and mostly perform better. Unfortunately, no memory usage information was provided. The SecreC 2 implementation spans over 2500 lines of code. It is heavily vectorized and performs rather complicated data reshaping steps.

6.6 Creating cryptographic challenges

The security of many cryptographic schemes is based on the hardness of a computational problem (discrete logarithm, integer factorization). In order to be able to select parameters for cryptographic schemes that guarantee a chosen level of security, the hardness of these problems has to be estimated. A method that has proved useful is to publish cryptographic challenges. For example, cryptographic challenges can be built for the factorization problem. The RSA factoring challenge, issued by RSA Labs in 1991, led to a much improved understanding of the strength of symmetric-key and public-key algorithms[2].

[2]http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge-faq.htm (April 2017)

Typically, to build a challenge someone generates an instance of the problem, and the challenge is to find a solution. In the case of integer factoring, a problem instance is a large number n and the expected solution is a pair of numbers $p, q > 1$ such that $p \times q = n$. In the case of the RSA problem p and q are large prime numbers.

However, for many computational problems it is hard to create an instance without knowing its solution. This means that the research community would have to trust the creator of the cryptographic challenge to not reveal any information about the solution and to generate the problem instance honestly. The creator himself would not be able to participate in the challenge. Instead of a single person generating a challenge, instances can be generated jointly by several parties using SMC.

Buchmann et al. [26] showed that SMC can be used to build challenges for the learning with errors (LWE) problem [102]—an important problem in lattice-based cryptography that is conjectured to be hard to solve. They implemented the challenge generation algorithm in SecreC 2 on Sharemind and generated instances for various parameters. The generation of problem instances took only a few minutes for the small cases but up to ten hours for the largest ones.

The algorithm spans just over 300 lines of code. Much of the code consists of comments and also a less efficient generation algorithm that used floating-point numbers. The parallelized generation algorithm itself, and the rest of the SecreC 2 application that also implements communication with the controller application, takes less than 200 lines of code. The client (controller) application was implemented in C++ and Python.

6.7 The SecreC code verifier

SecreC programs are meant for processing sensitive information. Thus, it is important that programs work as intended and do not reveal more sensitive information than planned. SecreC 2 already contains a basic information flow type system which enforces that data can only be revealed explicitly. However, many programs open data in the middle of execution and make branching decisions based on the public values. It is often not obvious that revealed intermediate values only leak the intended information.

During the FP7-funded PRACTICE project, INESC Porto developed the *Computation Specification Analysis* (CSA) tool for static analysis of SecreC 2 code. The tool can be used to specify and verify functional correctness and data leakage properties. The CSA tool is a largely independent look at the SecreC 2 language. The parser and type checker have been independently designed and developed. The tool builds on top of Dafny [82] and Boogie [3, 81] static program verification

Listing 6.2: Functional correctness property in CSA

```
//@ template <domain D>
//@ function D int sum(D int[[1]] xs)
//@ { size(xs) == 0 ? 0 : xs[0] + sum(xs[1:]) }

template <domain D>
D int sum(D int[[1]] vec)
//@ ensures \result == sum(vec);
{ ... }
```

Listing 6.3: Leakage property in CSA

```
template <domain D>
int leak(D int x)
//@ leakage requires public(x)
{ return declassify(x); }
```

toolchains. CSA is available in GitHub[3].

To support verification, SecreC 2 was extended with a specification language inspired by the Dafny language. Dafny is designed for verification of imperative programs and thus is well suited for SecreC 2. Functional and leakage properties are added to SecreC 2 programs in the form of comments. This way the program that is accepted by CSA is still be a legal SecreC 2 program and can be compiled without further modifications.

Let us look at two annotated SecreC 2 code examples. In Listing 6.2 we declare a summation procedure with an annotation that indicates the result is, in fact, the sum of the argument vector. In the example we have omitted the function body. Depending on the implementation it has to contain loop invariants and the user might have to prove some additional lemmas.

In Listing 6.3 we define a trivial function that simply declassifies its input. The procedure is annotated with a special `leakage` keyword stating when it is acceptable for a private input `x` to be made public. Without that annotation the function would be rejected by CSA. Using this functionality it is possible to enforce for functions the acceptable level of leakage. For example, a specialized efficient sorting function can enforce that it may only called if all possible element comparisons of the input array are allowed to leak.

[3]`https://github.com/haslab/SecreC` (April 2017)

CHAPTER 7

STATE OF THE ART

In this chapter we will give an overview of the history and the state of the art of programming languages in secure multi-party computation. We give a short overview of the more popular tools. However, we do not give the particulars of the theoretic advancements. We roughly classify the tools into ones based on garbled circuits, ones based on secret sharing, and hybrid tools. We will not give overview of SHAREMIND framework. We have already done so in Section 2.2. We compare most systems against either SECREC 2 or protocol DSL. Many of the systems that we cover share the goal of SECREC 2 to be intended for SMC application development by non-expert programmers. Protocol DSL aims making arithmetic secret-sharing based protocols easier to implement. This is rather unique goal that is not shared by many other systems.

7.1 Programming garbled circuits

7.1.1 Fairplay and SFDL

Fairplay [87] is a system that implements generic *secure function evaluation* (SFE). Fairplay is widely considered to be the first system for general-purpose secure two-party computation. It includes a high-level procedural language called *Secure Function Definition Language* (SFDL) that resembles Pascal or C and is tailored for SFE. The high-level language programs are compiled into Boolean circuits represented in a language called *Secure Hardware Definition Language* (SHDL). The circuits can be evaluated by two parties using Yao's garbled circuits technique.

An SFDL program consists of a sequence of global constant and type definitions followed by a sequence of function definitions. The language lacks recursion and unbounded loops in order to maintain obliviousness. Loops are only allowed if the number of iterations is known during compile time. SFDL supports standard if-statements that are implemented by evaluating both branches. Expressions of the language contain constants, variables, function calls and operators. The operators that are allowed include addition, subtraction, Boolean operations and standard comparisons. However, due to their cost, multiplication and division operators are not provided as primitives. The language also supports arrays but not accessing them with private indices. SFDL programs can be written using a graphical editor specifically designed for this purpose.

FairplayMP [7] is a generic system for secure multi-party computation that supplements Fairplay. The system also extends SFDL by allowing an unlimited number of participants. Participants are classified into three roles: input, computation and result parties. A single party can have multiple roles. The new language revision also allows accessing individual bits of a number with array-like notation. They introduced basic support for polymorphic functions. For example, it is possible to write a function that shifts any integer value regardless of its size.

In FairplayMP circuits are evaluated using an enhanced version of Beaver, Micali and Rogaway [5] (BMR) protocol that runs in a constant number of communication rounds. The protocol is based on Yao's two-party scheme but supports any number of participants.

SFDL has similar goals to SecreC. Both aim to be high-level algorithm specification languages. However, SFDL lacks mixed-mode computation and only has private data types. SFDL is also restricted to using garbled circuits whereas the choice of security scheme is not fixed in SecreC. SFDL 2.0 has limited form of polymorphism that allows for functions to be generic over any integer type. FairplayMP and SFDL 2.0 have been used to implement prototypes for second price auction and voting [7].

7.1.2 FastGC

FastGC is a set of techniques introduced by Huang et al. [57] for improving the running time and memory requirements of Yao's garbled circuit protocol. In previous implementations of Yao's protocol, the garbled circuit is fully generated and loaded into memory before circuit evaluation starts. This is not a problem for simple functions but even small non-trivial applications can require circuits with billions of gates. Huang et al. observed that it is unnecessary to generate and store the entire garbled circuit at once. By sorting the gates topologically and pipelining the process of circuit generation and evaluation they significantly improved overall efficiency and scalability. Their implementation never stores the entire garbled circuit. This effectively allows for an unlimited number of gates to be evaluated using a small memory footprint.

Their implementation allows programmers to construct applications in Java using a combination of high-level code for application specification and low-level code for constructing efficient circuits. Their framework includes a library of circuits optimized for garbled execution. High-level applications can be built by just composing these circuits. However, custom designed circuits for improved performance can be designed by programmers. For designing Boolean circuits no understanding of the cryptographic protocols for evaluating them is needed. FastGC framework has been used to implement privacy-preserving matrix factorization [93] that scales to ten thousand records. Matrix factorization can be basis for recommender systems. FastGC has been also use to build smartphone prototype applications for private set intersection and personal genetics [56].

7.1.3 CMBC-GC

CMBC-GC [55] is a tool that achieves secure two-party computation for ANSI C. It translates C programs into equivalent monolithic Boolean circuits and evaluates the circuits using Yao's protocol. CBMC attempts to minimize the size of the formulas and also optimizes the Boolean circuits to improve the efficiency of garbled circuit evaluation. Not all features of C are supported. CBMC-GC requires all loops and recursive call chains to be bounded by a constant. Floating-point computations are not supported and pointer arithmetic is also limited. Not all built-in primitive types of ANSI C are supported.

The authors of CMBC-GC claim that for a developer, programming an SMC system should be similar to programming an embedded processor or micro-controller. Developing SMC applications has to become a normal programming task in a standard programming language that has a compiler and other productivity tools, the same goal is shared by SecreC. They claim that the programming language should be standardized. They chose C for various reasons: closeness to the underlying hardware, and ubiquity. Many programmers can write C programs with ease, and a large code base is available. For example, Pullonen and Siim [98] modified an existing efficient fully IEEE 754 compliant software floating-point implementation SoftFloat[1] and compiled it to secure circuits using CMBC-GC.

CMBC-GC has similar goals to SecreC but the two taken very different approaches. The former adopts C directly while the latter only has C-like syntax but is otherwise very different language. One advantage of domain-specific approach is that SecreC allows for mixed-mode computations. As demonstrated Pullonen and Siim [98] the two systems can work together. CBMC-GC can be used to specify low-level operations and SecreC can be used to compose them to large applications. This combination of GC and secret sharing offers security again passive adversaries.

[1]`http://www.jhauser.us/arithmetic/SoftFloat.html` (April 2017)

7.1.4 PCF circuit format

In *portable circuit format* [69] (PCF) a secure function is represented as a program that computes the Boolean circuit representation of that function. The Boolean circuit is evaluated using Yao's garbled circuits technique. Their insight is that it is not necessary to unroll loops until the protocol needs to be run. The only restriction with loops is that they must not depend on secret values. The programmer is responsible for ensuring that there are no infinite loops. In PCF the circuit is not stored entirely and wires will be deleted from memory when they are no longer required. Essentially, this approach unifies the circuit pipelining of FastGC [57] with an optimizing compiler.

For testing, the authors of PCF used the LCC compiler [44] as a high-level front-end to their system. The LCC compiler translates the input C source code to a intermediate bytecode representation. Their back-end performs optimizations and translates the bytecode into PCF format. The PCF representation is also optimized. They have implemented constant propagation and dead gate elimination that, when combined, are shown to reduce the number of non-XOR gates by up to two times. PCF has been used to implement two-party functionality for integer matrix multiplication, modular exponentiation and graph isomorphism [69].

7.1.5 SCVM

The general-purpose secure computation implementations we have seen so far assume that the underlying functions are represented as circuits. Circuits are a sensible representation but typically programs use a von Neumann style *random-access machine* (RAM) model. Compiling such programs to efficient circuits can be challenging. Specifically, it is difficult to handle dynamic memory access where the memory location being read or written depends on private inputs. Program-to-circuit compilers typically make a copy of the entire memory object upon every private access to it. This results in a huge circuit when large objects are accessed.

SCVM [84] is an intermediate language for two-party RAM-model secure computation. It is the first automated approach for RAM-model SMC. The language supports accessing memory ia private indices by using *oblivious RAM* (ORAM). ORAM is a primitive that hides memory-access patterns.

The RAM-model has two main advantages over the circuit model. First, when performing repeated sublinear queries over a large dataset, the RAM model achieves sublinear amortized cost per query. Second, when performing one-time computation over a large dataset, it avoids the linear cost per private memory access.

SCVM is a quite low-level language supporting basic expressions, statements and arrays. The language does not feature any higher-level constructs like procedures. It has four information flow types: public, secure, and one local mode for each of the two participants. SCVM allows secret data to be revealed in the middle of execution. After declassification, the (public) control flow of the program may depend on the revealed secret values. The information flow type system of SCVM ensures that any program is secure in the semi-honest model if all its subroutines are secure in the semi-honest model.

The authors of SCVM also informally described how annotated C-like source language can be transformed into SCVM format. They built an automated compiler that does so and also integrates compile-time optimizations for improving performance. For instance, the compiler can identify parts of the program that can be safely executed locally by one or both of the parties. In addition, the compiler can detect memory accesses that do not depend on secret inputs. Using this information it is possible to avoid using ORAM when the access pattern is independent of sensitive inputs.

SCVM is intended to be the target language that higher-level secure programming languages are compiled to. Similar to SecreC it supports mixed-model computations but otherwise the languages have little in common.

7.1.6 ObliVM

ObliVM [85] is a programming framework for secure computation. It offers a domain-specific language, called ObliVM-lang, designed for compilation of programs into efficient representations suitable for secure computation. The high-level language is designed to allow programmers who are not familiar with secure computing or cryptography in general to write SMC applications. ObliVM-lang extends the SCVM language with function calls inside secret if-statements, native types, polymorphism, type-level natural numbers, and functions. Just like SCVM it uses ORAM to support accessing memory with secure indices.

The underlying primitive circuits can be specified directly in ObliVM-lang. Alternatively, primitives can still be implemented in the ObliVM framework and made available in the high-level language. Currently, ObliVM supports a semi-honest garbled circuits based back-end, but developers can implement customized special-purpose types and functions. The authors of ObliVM [85] claim that it would not be too hard to extend the system to support additional Boolean-circuit based back-ends such as GMW or FHE.

ObliVM adopts the pipelined circuit generation by Huang et al. [57]. The circuit is not generated entirely, only parts of it are. Thus, the required amount of working memory is drastically reduced compared to full-circuit based approaches.

SecreC and ObliVM both stride to make secure computation accessible to non-expert programmers. They share very similar goals. Major difference is that SecreC is security scheme independent very explicitly whereas ObliVM is Boolean-circuits based. Unlike ObliVM SecreC does not support ORAM natively but it is possible to implement ORAM in the language itself or via protection domain callbacks as done by Laud [74]. ObliVM has been used to implement various algorithms (Dijkstra's algorithm, minimum spanning tree, K-means) [85].

7.2 Programming protocols based on secret sharing

7.2.1 SMCL

The first imperative programming language for general secure computation was introduced by Nielsen and Schwartzbach in 2007 [92]. The language was dubbed SMCL for *Secure Multi-party Computation Language*. It is a procedural language syntactically similar to Java where various groups of clients interact with a server that is capable of performing secure computing with a client's shares. From the client's point of view it is interacting with a single computing engine and the fact that the server consists of multiple communicating physical machines is hidden. Code for groups of participants and the server is written in the same language and it is simple to emulate the concepts of input and output parties (Section 2.1.1) with this setup. The compiler translates SMCL code to Java.

Consider the problem of finding out which member of a group has the highest income. In this setting there is one group of participants and the server. The server may consist of three computing parties and the clients' group consists of any number of clients that each supply their salary information to the server in secret shared form. The server can be viewed as a single entity that, with the help of SMC, offers computing capabilities. The server side code iterates over all clients, receives the clients' income information through a channel, computes the maximum of the salaries, and then reveals to every client if they have the highest income or not. The client side code sends the salary information to the server and then waits for the server to reveal a Boolean indicating if they were the richest or not. Note that each server only learns a single share of each salary and will not learn any actual salary information.

Types in SMCL are split into two: secure types and public types. For example, secure integers are denoted with **`sint`** and public integers with **`int`**. The language allows secure values to be explicitly *opened* (revealed to the public, declassified). The language admits trace security, meaning that executions of a program on different private inputs look indistinguishable externally until a private value is opened.

Both SMCL and SecreC are intended for the same purpose, both are imperative languages and both allow for mixed-mode computation. In SMCL the code for input and output parties can be specified whereas SecreC only involves computing parties. In SecreC security level is a part of a type whereas in SMCL secure integers and public integers are two distinct data types. SecreC supports security scheme polymorphic functions. Notably, SMCL has been used in the implementation of the Danish sugar beet auction in 2008 [24, 91].

7.2.2 Launchbury's secure computing DSL

Launchbury et al. [78] developed a Haskell library that provides an embedded domain-specific language (EDSL) for programming SMC applications. Secure computation is based on three-party additive and XOR secret sharing. To improve efficiency, the EDSL supports SIMD parallelism and also task-level parallelism. The language provides operations for network communication, generating randomness, and computations with shares. The programmer can write protocols operating on secret shared values, and also leverage the full power of the host programming language.

Launchbury et al. detail an approach for optimizing secure SMC programs by moving to a symbolic representation where concrete types are replaced by symbolic counterparts. They transform the symbolic representation to one where all loops have been unrolled and lists have been eliminated. After that they group single multiplications into SIMD-style vectorized multiplications when data dependencies allow it. They applied this optimization to their implementation of AES but the ideas are general enough to be applicable elsewhere.

Our protocol DSL is quite similar in its goals to Launchbury's DSL. It could be considered natural advancement of ideas. We offer better safety guarantees via size-aware data type, party type system and security verification. We also have more advanced networking primitive and offer more aggressive optimizations. We compile to automatically vectorized efficient LLVM code. As a downside we have lost some expressive power because our language is not embedded in Haskell. Launchbury's DSL has been used for efficient SMC implementation of AES [78].

7.2.3 Mitchell's secure computing DSL

Mitchell et al. [88] have introduced a Haskell-based embedded domain-specific language for secure cloud computing. The language features primitive types, conditionals, standard functional language features, mutable state, a secrecy-preserving form of general recursion, and security type system that prevents control flow leakage. The language supports three back-ends. One of them is intended for debugging, allowing developers to easily test the code locally without any cryptographic protection. Two are intended for actual cloud deployments. One of these is based on secret sharing and the other on homomorphic encryption. Mixing the two security schemes is not natively supported by the language.

Mitchell's embedded DSL shares goals with SecreC language. Both aim to make SMC application development easier. However, this is where similarities end.

SecreC 2 is imperative language with more advanced type system that can be used with different security schemes. Same code can be shared across multiple different schemes. Mitchell's DSL has been used to implement an application that checks if an email address is present in a secret whitelist of emails [88].

7.2.4 PICCO

PICCO [118] is a system for converting programs written in a C-like language into secure implementations and running them in a distributed environment. The language preserves most C features and allows variables to be marked as private to be used in general-purpose secure computations. The secure implementation of compiled programs is based on Shamir's secret sharing, achieving information-theoretical security against passive adversaries. PICCO supports parallel execution of parts of the program either by explicit parallel loops or parallel execution of individual statements. PICCO also has a limited form of SIMD parallelism. For example, multiplication of private arrays operates pointwise. In [117] PICCO is extended to support pointers to private data and dynamic memory allocation.

Information flow security is enforced via the type system. Namely, conversions from private values to public values are not allowed, and branching over private Booleans is restricted. Private side effects, e.g., assignments to private variables, are permitted in functions called from the body of an if-statement with a private condition. Private values can be explicitly revealed with the `smcopen` function and the execution of the program can continue on the revealed public value. Public data can always be implicitly converted to private data.

SecreC and PICCO share similar goals. Both are intended to abstract away the underlying secure computation details while still enabling programmers to write performance efficient applications. PICCO takes the concurrency approach while SecreC is strictly limited to SIMD parallelism. Both offer basic data-flow security guarantees via type system. PICCO is not security scheme agnostic. PICCO has been used to implement various algorithms (matrix multiplication, merge sort, AES) [118], prototype fingerprint matching application, and a shift-reduce parser for parsing private data [117] .

7.2.5 Wysteria

Wysteria [100, 99] is a high-level functional programming language for writing SMC applications. It features higher-order functions, let-expressions, tuples, sum types, and primitive values such as integers and arrays. Wysteria programs may involve an arbitrary number of parties. The set of parties (and its size) may be determined dynamically during computation. Programs are compiled to Boolean

```
let a   =par({Alice})=      read() in
let b   =par({Bob})=        read() in
let out =sec({Alice, Bob})= a[Alice] > b[Bob] in
out
```

Figure 7.1: Mixed-mode computation in Wysteria (reproduced [100])

circuits that are securely executed by an underlying engine using the GMW protocol [50].

It is possible to write programs in Wysteria in mixed-mode style, interleaving local and private computation. Namely, the programs operate in a combination of parallel and secure modes. Parallel mode indicates that one or more parties are performing local computations in parallel. Secure mode indicates secure computations occurring jointly among some subset of parties. For example, in Figure 7.1 three variables are defined. Variables `a` and `b` are computed in parallel mode, by `Alice` and `Bob` respectively. The result `out` is securely jointly computed, checking if `a` is greater than `b`.

Wysteria provides secret shares and *wire bundles* as first-class objects. Wire bundles are used to represent the public inputs and outputs of secure computations. They are similar to secret shares in that each party has a local copy. With secret shares the copies are combined to reconstruct a single value but with wire bundles each copy is its own individual value. A single party's view of a wire bundle is his own value, while the shared view represents all possible values. A secure computation, having the shared view, may iterate over the contents of a bundle. The length of wire bundles may be unspecified during compile time.

Wysteria features a strong type system. The language is dependently typed in the sense that functions take the participants as input, and the types for wire bundles and shares directly identify the parties involved. The type system also limits from which contexts a function can be called. For example, within a secure computation it is not possible to call a function that computes in parallel mode, while the reverse is possible. The type system also ensures that shares are used properly. For example, shares of different objects cannot be combined.

The mixed-mode computation can be achieved in SecreC to a degree. Public and private computations can be interleaved arbitrarily and more complicated schemes can be emulated by using protection domains for local computations. Public domain is SecreC corresponds to the parallel mode where all parties executing the program are involved. However, SecreC does not directly support dynamically choosing which parties to compute with. The protection domain is picked statically. The similar style could in theory be emulated with security scheme specific operations.

Purposes of SecreC and Wysteria diverge slightly. One of the primary goals of SecreC is to abstract away the detail of SMC and offer programmers the illusion of executing regular single threaded code with straightforward control flow. In that sense Wysteria is a lower level language and makes the concepts of participants and the data they keep explicit. SecreC does not have the concept of a participant and is intended to be security scheme agnostic and even support, for example, hardware based secure computation. Wysteria on the other hand explicitly supports n-party computations and does not allow the programmer to swap security schemes in the language itself. Any security scheme that Wysteria supports needs to be able to handle arbitrary number of parties. Wysteria has been used to implement various oblivious algorithms [100] (private set intersection, median, nearest neighbors) and secure application for card dealing [99].

7.3 Programming protocols with hybrid tools

7.3.1 TASTYL

TASTY [53] is a tool for describing, generating, executing, benchmarking, and comparing secure two-party computation protocols. The user provides a high-level description of the secure computation in a domain-specific language. TASTYL is a high-level language for describing secure computations on encrypted data, allowing to abstract the details of the underlying cryptographic protocols.

The authors note that homomorphic encryption and garbled circuits have different performance characteristics. Either one could be more efficient depending on the application. For example, integer multiplication is faster with homomorphic encryption but garbled circuits offer a more efficient comparison operation. Thus, for some applications using both yields a more efficient solution than using either one separately. TASTY supports both homomorphic encryption and Yao's garbled circuits and allows conversions between the two schemes. The schemes can both be used in the same application to improve efficiency.

TASTYL is a subset of Python programming language and as such is quite high-level making it comparable to SecreC. TASTYL is specific to two party computation and a programmer has to explicitly code in client-server setting. SecreC aims to hide whether a security scheme requires distributed computation.

7.3.2 L1

L1 [103] is an intermediate language for secure computation compilers. It enables the implementation of protocols potentially mixing several general secure computing schemes. The language supports many schemes such as secret sharing, homomorphic encryption, garbled circuits, but also task specific secure computing protocols.

L1 is supposed to be used as an intermediate language for secure computation but could also support the writing of high-level applications. It is syntactically similar to Java or C. The language features explicit network communication and parallel execution of basic blocks that is explicitly controlled by the programmer.

In [66] Kerschbaum, Schneider and Schröpfer proposed a way of automatically selecting a protocol for each operation. They claim that the selection problem is so complicated and large that a developer is unlikely to manually make the optimal selection. They achieve improved performance over a pure garbled circuits based implementation.

7.3.3 ABY

ABY [40] is a mixed-protocol secure two-party computation framework that combines computation schemes based on arithmetic sharing, Boolean sharing, and Yao's garbled circuits. The framework allows to pre-compute almost all cryptographic operations and provides efficient conversions between the different representations. The task of specifying which computation is executed with which scheme is left to the user.

ABY supports converting between the three different types of representations and performing standard operations such as addition, multiplication, comparison, and various bitwise operations. For arithmetic sharing the framework uses protocols based on multiplication triples, and for Boolean sharing the GMW [50] protocol.

The ABY framework abstracts from the underlying secure computation protocols. Variables are either public or secret shared between the two parties. The authors of ABY claim that high-level languages can be compiled into their framework and it can be used as a back-end for other secure computation tools. ABY has been used to develop modular exponentiation algorithm and example applications for private set intersection and biometric matching [40].

CONCLUSION

This thesis studies how secure multi-party computation technology can benefit from domain-specific programming languages. High-level applications that use multi-party computation have requirements that general-purpose programming languages do not satisfy. Implementing secure multi-party protocols poses unique technical problems. These problems can be alleviated with specialized languages that both simplify development and increase trust in correctness and security.

The main result of this thesis is the implementation and formalization of two programming languages. The first language, called SecreC 2, is intended for secure multi-party application and algorithm development. Its goal is to make secure multi-party applications with good performance easy to develop for programmers who are not security experts. The second domain-specific language is intended for low-level secure computation protocol development. Its aim is to make protocols easier to develop and to increase the trust that protocols are correct and secure.

For the SecreC 2 language we give a very informal overview that is meant for the language those who with to learn the language. We rely heavily on examples and show how to develop code that is reusable and performs well. In addition the informal overview we formalize the core of the SecreC 2 language and describe its type system and its dynamic behavior. We show that if the underlying protocols invoked by SecreC 2 are universally composable, then the whole program is secure. We describe how to translate SecreC 2 programs to a non-polymorphic form and show that the translation preserves the programs' behavior and therefore their security.

For the domain-specific protocol language we also give an informal overview. This is mostly intended for curious readers, the users of the language, and anyone who is interested how three-party additive protocols work. We show that the language is expressive enough to implement all integer operations and also complicated floating-point arithmetic. We also show that protocols developed in this language perform significantly better than our previous protocols. In some cases, we show performance improvement of over two orders of magnitude.

Finally, we give an overview of applications and prototypes that have been developed using SecreC 2. We can safely state that SecreC 2 is the most used secure multi-party application development language with the largest reusable standard library. Most of the applications and prototypes benefit from the protocol language as well because it has now been used to implement the vast majority of SHAREMIND protocols.

Bibliography

[1] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases. pp. 487–499. VLDB ’94, Morgan Kaufmann Publishers Inc. (1994)

[2] Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 805–817. CCS ’16, ACM (2016)

[3] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387. Springer (2005)

[4] Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. pp. 307–314. AFIPS ’68 (Spring), ACM (1968)

[5] Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing. pp. 503–513. STOC ’90, ACM (1990)

[6] Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Annual International Cryptology Conference. pp. 420–432. Springer (1991)

[7] Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS ’08: Proceedings of the 15th ACM conference on Computer and communications security. pp. 257–266. ACM (2008)

[8] Blakley, G.: Safeguarding cryptographic keys. In: Proceedings of the 1979 AFIPS National Computer Conference. pp. 313–317. AFIPS Press, Monval, NJ, USA (1979)

[9] Bogdanov, D., Kamm, L., Laur, S., Sokk, V.: Rmind: a Tool for Cryptographically Secure Statistical Analysis. IEEE Transactions on Dependable and Secure Computing PP(99) (2016)

[10] Bogdanov, D.: How to securely perform computations on secret-shared data. Master's thesis, University of Tartu (2007)

[11] Bogdanov, D.: Sharemind: programmable secure computations with practical applications. Ph.D. thesis, University of Tartu (February 2013)

[12] Bogdanov, D., Emura, K., Jagomägis, R., Kanaoka, A., Matsuo, S., Willemson, J.: A Secure Genetic Algorithm for the Subset Cover Problem and Its Application to Privacy Protection. In: 8th IFIP International Workshop on Information Security Theory and Practice (WISTP). Information Security Theory and Practice. Securing the Internet of Things, vol. 8501, pp. 108–123. Springer (Jun 2014)

[13] Bogdanov, D., Jagomägis, R., Laur, S.: Privacy-preserving Histogram Computation and Frequent Itemset Mining with Sharemind. Tech. Rep. T-4-8, Cybernetica, `http://research.cyber.ee/`. (2009)

[14] Bogdanov, D., Jagomägis, R., Laur, S.: A Universal Toolkit for Cryptographically Secure Privacy-Preserving Data Mining. In: Intelligence and Security Informatics - Pacific Asia Workshop, PAISI'12, Kuala Lumpur, Malaysia, May 29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7299, pp. 112–126. Springer (2012)

[15] Bogdanov, D., Jõemets, M., Siim, S., Vaht, M.: How the Estonian Tax and Customs Board Evaluated a Tax Fraud Detection System Based on Secure Multi-party Computation. In: Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8975, pp. 227–234. Springer (2015)

[16] Bogdanov, D., Jõemets, M., Siim, S., Vaht, M.: Privacy-preserving tax fraud detection in the cloud with realistic data volumes. Tech. Rep. T-4-24, Cybernetica AS, `http://research.cyber.ee/`. (2016)

[17] Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., Talviste, R.: Students and Taxes: a Privacy-Preserving Study Using Secure Computation. PoPETs 2016(3), 117–135 (2016)

[18] Bogdanov, D., Laud, P., Laur, S., Pullonen, P.: From input private to universally composable secure multi-party computation primitives. In: IEEE 27th

Computer Security Foundations Symposium, CSF 2014. pp. 184–198. IEEE (July 2014)

[19] Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic language for privacy-preserving applications. In: PETShop'13, Proceedings of the 2013 ACM Workshop on Language Support for Privacy-Enhancing Technologies, Co-located with CCS 2013, November 4, 2013, Berlin, Germany. pp. 23–26 (2013)

[20] Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014. pp. 53–65 (2014)

[21] Bogdanov, D., Laur, S., Talviste, R.: A Practical Analysis of Oblivious Sorting Algorithms for Secure Multi-party Computation. In: Proceedings of the 19th Nordic Conference on Secure IT Systems, NordSec 2014, Lecture Notes in Computer Science, vol. 8788, pp. 59–74. Springer (2014)

[22] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008)

[23] Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. Int. J. Inf. Sec. 11(6), 403–418 (2012)

[24] Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers. Lecture Notes in Computer Science, vol. 5628, pp. 325–343. Springer (2009)

[25] Boneh, D., Goh, E.J., Nissim, K.: Evaluating 2-DNF formulas on ciphertexts. In: Theory of Cryptography Conference. pp. 325–341. Springer (2005)

[26] Buchmann, J.A., Büscher, N., Göpfert, F., Katzenbeisser, S., Krämer, J., Micciancio, D., Siim, S., van Vredendaal, C., Walter, M.: Creating cryptographic challenges using multi-party computation: The LWE challenge. In: Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography, AsiaPKC@AsiaCCS, Xi'an, China, May 30 - June 03, 2016. pp. 11–20. ACM (2016)

[27] Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: USENIX Security Symposium. pp. 223–239. Washington, DC, USA (2010)

[28] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS. pp. 136–145 (2001)

[29] Canetti, R., Cohen, A., Lindell, Y.: A simpler variant of universally composable security for standard multiparty computation. In: Annual Cryptology Conference. pp. 3–22. Springer (2015)

[30] Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6052, pp. 35–50. Springer (2010)

[31] Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: Proceedings of the twentieth annual ACM symposium on Theory of computing. pp. 11–19. ACM (1988)

[32] Cramer, R., Damgård, I., Nielsen, J.B.: Secure Multiparty Computation and Secret Sharing. Cambridge University Press (July 2015)

[33] Cramer, R., Gennaro, R., Schoenmakers, B.: A secure and optimally efficient multi-authority election scheme. European transactions on Telecommunications 8(5), 481–490 (1997)

[34] Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 207–212. ACM (1982)

[35] Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous Multiparty Computation: Theory and Implementation. In: Public Key Cryptography. Lecture Notes in Computer Science, vol. 5443, pp. 160–179. Springer (2009)

[36] Damgård, I., Jurik, M.: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In: International Workshop on Public Key Cryptography. pp. 119–136. Springer (2001)

[37] Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: CRYPTO. Lecture Notes in Computer Science, vol. 2729, pp. 247–264. Springer (2003)

[38] Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Advances in Cryptology–CRYPTO 2012, pp. 643–662. Springer (2012)

[39] Demmler, D., Dessouky, G., Koushanfar, F., Sadeghi, A.R., Schneider, T., Zeitouni, S.: Automated synthesis of optimized circuits for secure computation. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1504–1517. ACM (2015)

[40] Demmler, D., Schneider, T., Zohner, M.: ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In: NDSS (2015)

[41] Dimitrov, V., Kerik, L., Krips, T., Randmets, J., Willemson, J.: Alternative implementations of secure real numbers. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 553–564. CCS '16, ACM (2016)

[42] Ducas, L., Micciancio, D.: FHEW: bootstrapping homomorphic encryption in less than a second. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 617–640. Springer (2015)

[43] Fournet, C., Guernic, G.L., Rezk, T.: A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In: Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009. pp. 432–441 (2009)

[44] Fraser, C.W., Hanson, D.R.: A retargetable C compiler: design and implementation. Addison-Wesley Longman Publishing Co., Inc. (1995)

[45] Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31(4), 469–472 (1985)

[46] Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009)

[47] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. pp. 169–178. ACM (2009)

[48] Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 129–148. Springer (2011)

[49] Giannakopoulos, Y., Guanciale, R., Kamm, L., Laud, P., Pankova, A., Pettai, M., Pullonen, P., Siim, S., Tselekounis, Y.: Scientific Progress Analysis and Recommendations (July 2015), http://usable-security.eu/workpackages-and-reports/wp5-scientific-coordination/d523-scientific-progress-analysis-and-recommendations.html, uaESMC Deliverable 5.2.3

[50] Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: STOC. pp. 218–229. ACM (1987)

[51] Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers. pp. 202–216 (2012)

[52] Hauser, J.: Berkeley SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html, accessed: 2017-04-19

[53] Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10. pp. 451–462. ACM (2010)

[54] Hoare, C.A.: Algorithm 65: find. Communications of the ACM 4(7), 321–322 (1961)

[55] Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. pp. 772–783. ACM (2012)

[56] Huang, Y., Chapman, P., Evans, D.: Privacy-preserving applications on smartphones. In: 6th USENIX Workshop on Hot Topics in Security, HotSec'11, San Francisco, CA, USA, August 9, 2011 (2011)

[57] Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: USENIX Security Symposium. vol. 201 (2011)

[58] Jagomägis, R.: A programming language for creating privacy-preserving applications. Bachelor's thesis. University of Tartu (2008)

[59] Jagomägis, R.: SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu (2010)

[60] Kamm, L.: Privacy-preserving statistical analysis using secure multi-party computation. Ph.D. thesis, University of Tartu (2015), http://hdl.handle.net/10062/45343

[61] Kamm, L., Bogdanov, D., Laur, S., Vilo, J.: A new way to protect privacy in large-scale genome-wide association studies. Bioinformatics 29(7), 886–893 (2013), http://bioinformatics.oxfordjournals.org/content/29/7/886.abstract

[62] Kamm, L., Willemson, J.: Secure Floating Point Arithmetic and Private Satellite Collision Analysis. International Journal of Information Security pp. 1–18 (2014)

[63] Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure MPC with dishonest majority. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. pp. 549–560. ACM (2013)

[64] Kerik, L., Laud, P., Randmets, J.: Optimizing MPC for robust and scalable integer and floating-point arithmetic. In: Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. pp. 271–287 (2016)

[65] Kerschbaum, F.: Automatically optimizing secure computation. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 703–714. ACM (2011)

[66] Kerschbaum, F., Schneider, T., Schröpfer, A.: Automatic protocol selection in secure two-party computations. In: Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings. pp. 566–584 (2014)

[67] Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. 22(8), 786–793 (Aug 1973)

[68] Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: International Colloquium on Automata, Languages, and Programming. pp. 486–498. Springer (2008)

[69] Kreuter, B., Shelat, A., Mood, B., Butler, K.R.B.: PCF: A portable circuit format for scalable two-party secure computation. In: Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013. pp. 321–336. USENIX Association (2013), https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/kreuter

[70] Kreuter, B., Shelat, A., Shen, C.H.: Billion-gate secure computation with malicious adversaries. In: Proceedings of the 21st USENIX conference on Security symposium. pp. 285–300. USENIX Association (2012)

[71] Krips, T., Willemson, J.: Hybrid model of fixed and floating point numbers in secure multiparty computations. In: Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings. pp. 179–197 (2014)

[72] Ladner, R.E., Fischer, M.J.: Parallel prefix computation. Journal of the ACM (JACM) 27(4), 831–838 (1980)

[73] Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004), http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=9012

[74] Laud, P.: Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. PoPETs 2015(2), 188–205 (2015)

[75] Laud, P., Pankova, A.: Privacy-preserving frequent itemset mining for sparse and dense data. Cryptology ePrint Archive, Report 2015/671 (2015), http://eprint.iacr.org/2015/671

[76] Laud, P., Pankova, A., Pettai, M., Randmets, J.: Specifying Sharemind's arithmetic black box. In: PETShop'13, Proceedings of the 2013 ACM Workshop on Language Support for Privacy-Enhancing Technologies, Co-located with CCS 2013, November 4, 2013, Berlin, Germany. pp. 19–22 (2013)

[77] Laud, P., Randmets, J.: A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1492–1503 (2015)

[78] Launchbury, J., Diatchki, I.S., DuBuisson, T., Adams-Moran, A.: Efficient lookup-table protocol in secure multiparty computation. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012. pp. 189–200 (2012)

[79] Laur, S., Talviste, R., Willemson, J.: From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In: Applied Cryptography and Network Security, Lecture Notes in Computer Science, vol. 7954, pp. 84–101. Springer (2013)

[80] Laur, S., Willemson, J., Zhang, B.: Round-Efficient Oblivious Database Manipulation. In: Proceedings of the 14th International Conference on Information Security. ISC'11. pp. 262–277 (2011)

[81] Leino, K.R.M.: This is boogie 2. Manuscript KRML 178, 131 (2008)

[82] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)

[83] Lewis, J.: Cryptol: specification, implementation and verification of high-grade cryptographic applications. In: FMSE. p. 41. ACM (2007)

[84] Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient ram-model secure computation. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 623–638. IEEE Computer Society (2014), `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6954656`

[85] Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: ObliVM: A programming framework for secure computation. In: 2015 IEEE Symposium on Security and Privacy. pp. 359–376. IEEE (2015)

[86] Malka, L., Katz, J.: Vmcrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584 (2010), `http://eprint.iacr.org/`

[87] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay—a secure two-party computation system. In: SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium. pp. 20–20. USENIX Association, Berkeley, CA, USA (2004)

[88] Mitchell, J.C., Sharma, R., Stefan, D., Zimmerman, J.: Information-flow control for programming on encrypted data. In: 2012 IEEE 25th Computer Security Foundations Symposium. pp. 45–60. IEEE (2012)

[89] de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 337–340 (2008)

[90] Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: Proceedings of the 1st ACM conference on Electronic commerce. pp. 129–139. ACM (1999)

[91] Nielsen, J.D.: Languages for secure multiparty computation and towards strongly typed macros. Ph.D. thesis, PhD thesis, University of Aarhus, Denmark (2009)

[92] Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security. PLAS'07. pp. 21–30. ACM (2007)

[93] Nikolaenko, V., Ioannidis, S., Weinsberg, U., Joye, M., Taft, N., Boneh, D.: Privacy-preserving matrix factorization. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 801–812. ACM (2013)

[94] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT. pp. 223–238 (1999)

[95] Parberry, I.: The pairwise sorting network. Parallel Processing Letters 2(02n03), 205–211 (1992)

[96] Parno, B., McCune, J.M., Perrig, A.: Bootstrapping trust in commodity computers. In: 2010 IEEE Symposium on Security and Privacy. pp. 414–429. IEEE (2010)

[97] Pettai, M., Laud, P.: Automatic Proofs of Privacy of Secure Multi-Party Computation Protocols Against Active Adversaries. In: 2015 IEEE 28th Computer Security Foundations Symposium (CSF 2015) (2015)

[98] Pullonen, P., Siim, S.: Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations. In: Financial Cryptography and Data Security - FC 2015 Workshops, BITCOIN, WAHC and Wearable 2015, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8976, pp. 172–183. Springer (2015)

[99] Rastogi, A.: Language-based Techniques for Practical and Trustworthy Secure Multi-party Computations. Ph.D. thesis, University of Maryland, College Park (2016)

[100] Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: A programming language for generic, mixed-mode multiparty computations. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 655–670. IEEE Computer Society (2014), http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6954656

[101] Rastogi, A., Mardziel, P., Hicks, M., Hammer, M.A.: Knowledge inference for optimizing secure multi-party computation. In: Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013. pp. 3–14 (2013)

[102] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) 56(6), 34 (2009)

[103] Schröpfer, A., Kerschbaum, F., Mueller, G.: L1 - An Intermediate Language for Mixed-Protocol Secure Computation. In: Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference. COMPSAC'11. pp. 298–307. IEEE Computer Society (2011)

[104] Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)

[105] Shin, S., Porras, P.A., Yegneswaran, V., Fong, M.W., Gu, G., Tyson, M.: Fresco: Modular composable security services for software-defined networks. In: NDSS (2013)

[106] Sokk, V.: An improved type system for a privacy-aware programming language and its practical applications. Master's thesis, Institute of Computer Science, University of Tartu (2016)

[107] Talviste, R.: Deploying secure multiparty computation for joint data analysis—a case study. Master's thesis, Institute of Computer Science, University of Tartu (2011)

[108] Talviste, R.: Applying Secure Multi-party Computation in Practice. Ph.D. thesis, University of Tartu (2016)

[109] Tamrakar, S., Liu, J., Paverd, A., Ekberg, J., Pinkas, B., Asokan, N.: The circle game: Scalable private membership test using trusted hardware. CoRR abs/1606.01655 (2016), http://arxiv.org/abs/1606.01655

[110] Turban, T.: Secure Multi-Party Computation Protocol Suite Inspired by Shamir's Secret Sharing Scheme. Master's thesis, Institute of Computer Science, University of Tartu (2014)

[111] Vaht, M.: The Analysis and Design of a Privacy-Preserving Survey System. Master's thesis, Institute of Computer Science, University of Tartu (2015)

[112] Vytiniotis, D., Peyton Jones, S., Schrijvers, T., Sulzmann, M.: OutsideIn(X) Modular type inference with local assumptions. Journal of Functional Programming 21(4-5), 333–412 (2011)

[113] Yao, A.C.C.: Protocols for secure computations (extended abstract). In: FOCS. pp. 160–164. IEEE (1982)

[114] Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: FOCS. pp. 162–167 (1986)

[115] Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole - reducing data transfer in garbled circuits using half gates. In: Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II. pp. 220–250 (2015)

[116] Zaki, M.J.: Scalable algorithms for association mining. IEEE Transactions on Knowledge and Data Engineering 12(3), 372–390 (2000)

[117] Zhang, Y., Blanton, M., Almashaqbeh, G.: Implementing support for pointers to private data in a general-purpose secure multi-party compiler. arXiv preprint arXiv:1509.01763 (2015)

[118] Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. pp. 813–826. ACM (2013)

ACKNOWLEDGEMENTS

During my doctoral studies I have been supported by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science (EXCS), by the European Social Fund through the Estonian Doctoral School in Information and Communication Technology (IKTDK) and by the Software Technologies and Applications Competence Centre (STACC). I have also received support from the Estonian Research Council through project IUT27-1.

Compiler development is a daunting task and I have received invaluable support from my colleagues in Cybernetica AS. The SecreC 2 compiler was developed from privacy analysis the framework originally by Jaak Ristioja. Later, Ville Sokk has contributed significantly to the compiler. During the development of the protocol DSL compiler I have received help from Madis Janson, Martin Pettai and Karl Tarbe. The protocol DAG optimizer tool was developed by Alisa Pankova and the additive three-party protocol set was developed by Liisi Kerik.

I am very grateful to both of my supervisors. Peeter Laud has guided me through my doctoral studies, has helped me in both writing and in the theoretical sides of my work. Without Varmo Vene I would not have been introduced to the world of programming language design and theory, and I would never have had the opportunity to research such an interesting topic. My gratitude extends to the whole Sharemind team. Dan Bogdanov has supervised me unofficially and provided me with invaluable guidance during my work and studies. Liina Kamm gave me extremely helpful and very thorough feedback on my writing.

Finally, I would like to express my deepest gratitude to my family who have been very supportive of my academic career and Liisi Kerik who has supported me throughout this long and challenging writing process.

KOKKUVÕTE (SUMMARY IN ESTONIAN)

PROGRAMMEERIMISKEELED TURVALISE ÜHISARVUTUSE RAKENDUSTE ARENDAMISEKS

Turvaline ühisarvutus on krüptograafiline tehnoloogia, mis lubab mitmel sõltumatul osapoolel koos andmeid töödelda neis olevaid saladusi avalikustamata. Kui andmed on esitatud krüpteeritud kujul, tähendab see, et neid ei avata arvutuse käigus kordagi. Turvalise ühisarvutuse teoreetilised konstruktsioonid on teada olnud juba alates kaheksakümnendatest, kuid esimesed praktilised teostused ja rakendused, mis päris andmeid töötlesid, ilmusid alles veidi enam kui kümme aastat tagasi. Nüüdseks on turvalist ühisarvutust kasutatud mitmetes praktilistes rakendustes ning sellest on kujunenud oluline andmekaitsetehnoloogia.

Et turvaline ühisarvutus oleks paremini praktikas kasutatav, peab tehnoloogia ise ning olemasolevad raamistikud veel palju arenema. Probleeme on kasutusmugavusega, jõudlusega ning võimekusega töödelda suurtes kogustes andmeid. Rakendusi on endiselt võimelised arendama ainult krüptograafiaeksperdid ning tehnoloogia on liiga aeglane paljude praktiliste probleemide lahendamiseks.

Meie nägemus on, et hea jõudlusega turvalise ühisarvutuse rakendused peaksid olema lihtsalt arendatavad tavalise programmeerija jaoks, kes ei ole selle ala ekspert. Efektiivsed programmid ei tohiks olla liiga keerulised kirjutada. Käesoleva töö üks eesmärk on lihtsustada ühisarvutust kasutavate rakenduste arendamist. Selleks esitleme uut kõrgtaseme programmeerimiskeelt SecreC 2, mille on kavandanud ning teostanud töö autor.

Turvalise ühisarvutuse rakendusi arendatakse üldiselt valdkonnaspetsiifilisi programmeerimiskeeli kasutades, sest tavalised üldlevinud programmeerimiskeeled ei sobi selleks eesmärgiks kuigi hästi. Ühisarvutusrakendustes on väga oluline täpselt kontrollida, millised andmed avalikustatakse ja millised tuleb rangelt salajas hoida. Mitte mingil juhul ei tohi avalikustada andmeid, mida programmeerija ei ole ilmutatud kujul tahtnud avalikustada. Selline infovookontroll saavutatakse tavaliselt keele tüüpide tasemel: andmed on rangelt klassifitseeritud avalikeks ning salajasteks. Avalikke andmeid tohib alati vaikimisi teisendada salajaseks, aga vastupidine võib juhtuda ainult ilmutatud kujul. Mõnede üldlevinud keelte tüübisüsteemid on küll piisavalt võimsad, et selliseid infovookitsendusi väljendada, kuid nende kasutamine sellel eesmärgil ei ole kuigi mugav.

Veel üks oluline aspekt, milles turvalise ühisarvutuse rakendused erinevad tavalistest rakendustest, on jõudlus. Isegi parimad skeemid on endiselt suurusjärkudes aeglasemad avalikust arvutusest. Hea jõudlus on paljude skeemide korral saavutatav ainult kui kasutada paralleelsust sageli ning ka väga lihtsate arvutuste juures. Ühisarvutuse jaoks mõeldud programmeerimiskeeled peavad selle vajadusega arvestama ning operatsioonid ise peavad samuti olema efektiivsed. Kord korda kiirem tehe võib tähendada, et rakendus võtab mitmeid päevi vähem aega.

Turvalise ühisarvutuse platvorme saab vaadelda kui hajusarvuteid, mis pakuvad instruktsioone, mida kasutajad saavad välja kutsuda üldise arvutuse täitmiseks. Instruktsioonide jõudluse on äärmiselt oluline. Nende ühisarvutusraamistike jaoks, mis pakuvad suures koguses spetsialiseeritud primitiivoperatsioone, põhjustab see arendus- ja hooldusprobleeme. Sellist agressiivselt optimeeritud madalatasemelist võrgukoodi on keeruline ja veaohtlik teostada. Raske on kontrollida, kas operatsioonid on korrektselt teostatud ning ega nad ei sisalda turvavigu. Käesolevas töös esitleme uut valdkonnaspetsiifilise programmeerimiskeelt turvalise ühisarvutuse madalataseme protokollide teostamiseks. Selle keele peamine eesmärk on protokollide kirjutamist lihtsustada, aga lisaks ka tõsta jõdlust ning suurendada usaldust, et protokollid on korrektsed ja turvalised.

Väidame, et valdkonnaspetsiifiliste programmeerimiskeelte kasutamine võimaldab ehitada turvalise ühisarvutuse rakendusi ja raamistikke, mis on samaaegselt lihtsasti kasutatavad, hea jõudlusega, hooldatavad, usaldatavad ja võimelised suuri andmemahtusid töötlema. Töö autor on kavandanud ning teostanud kaks uut programmeerimiskeelt, mis on sammuks selle eesmärgi saavutamise suunas.

Esimene programmeerimiskeel on turvalise ühisarvutuse algoritmide ja rakenduste arendamiseks mõeldud kõrgtaseme keel SecreC 2. Käesolevas töös anname sellest keelest mitteformaalse paljude näidetega ülevaate, mis on mõeldud lugejatele, kes on huvitatud keele õppimisest või lihtsalt sellega tutvumisest. Lisaks kirjeldame keele tüübisüsteemi ja dünaamilist käitumist formaalselt ning arutleme keele turvalisuse teemal. Anname ülevaate kõikidest rakendustest, mis kasutavad SecreC 2 keelt. Näitame, et SecreC 2 lihtsustab rakenduste arendamist, võimaldab kirjutada hea jõudlusega programme ning saab hakkama suurte andmemahtudega. Saame julgelt väita, et SecreC 2 on enimkasutatav turvalise ühisarvutuse jaoks mõeldud programmeerimiskeel.

Teine programmeerimiskeel on mõeldud turvalise ühisarvutuse protokollide arendamiseks. Ka sellest keelest anname palju koodinäiteid sisaldava mitteformaalse ülevaate, kus esitame täisarvulised tehted kolme osapoolega aditiivse ühissalastuse jaoks. Ülevaade on mõeldud neile, kes soovivad tutvuda keele või protokollidega. Lisaks kirjeldame formaalselt keele tüübisüsteemi ning dünaamilist käitumist. Anname ülevaate keele vaheesitusest ning näitame, kuidas protokollide turvalisust automaatselt kontrollida. Analüüsime põhjalikult protokollide keeles teostatud tehete jõudlust ja näitame, et võrreldes meie eelmise teostusega kasvab paljude tehete jõudlus mitu suurusjärku. Meie uues keeles kirjutatud protokollid on kergesti hallatavad, nende kood on kompaktne ja väljendab selgelt programmeerija tahet ning teostatud protokollid on turvalised.

LIST OF ORIGINAL PUBLICATIONS

1. Laud, P., Pankova, A., Pettai, M., Randmets, J.: Specifying Sharemind's arithmetic black box. In: PETShop'13, Proceedings of the 2013 ACM Workshop on Language Support for Privacy-Enhancing Technologies, Co-located with CCS 2013, November 4, 2013, Berlin, Germany. pp. 19–22 (2013).

2. Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic language for privacy-preserving applications. In: PETShop'13, Proceedings of the 2013 ACM Workshop on Language Support for Privacy-Enhancing Technologies, Co-located with CCS 2013, November 4, 2013, Berlin, Germany. pp. 23–26 (2013).

3. Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014. pp. 53–65 (2014).

4. Laud, P., Randmets, J.: A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1492–1503 (2015).

5. Kerik, L., Laud, P., Randmets, J.: Optimizing MPC for robust and scalable integer and floating-point arithmetic. In: Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. pp. 271–287 (2016).

CURRICULUM VITAE

Personal data

Name	Jaak Randmets
Birth	March 12th, 1986 Kuressaare, Estonia
Citizenship	Estonian
Languages	Estonian, English
E-mail	jaak.randmets@cyber.ee

Education

2013–	University of Tartu, Ph.D. candidate in Computer Science
2008–2012	University of Tartu, M.Sc. in Computer Science
2005–2008	University of Tartu, B.Sc. in Computer Science
2002–2005	Kuressaare Gümnaasium, secondary education
1993–2002	Kuressaare Gümnaasium, primary education

Employment

2013–	Cybernetica AS, junior researcher
2011–2013	Cybernetica AS, programmer
2010–2011	Software Technology and Applications Competence Centre (STACC), software developer

ELULOOKIRJELDUS

Isikuandmed

Nimi	Jaak Randmets
Sünniaeg ja -koht	12. märts 1986 Kuressaare, Eesti
Kodakondsus	eestlane
Keelteoskus	eesti, inglise
E-post	jaak.randmets@cyber.ee

Haridustee

2013–	Tartu Ülikool, informaatika doktorant
2008–2012	Tartu Ülikool, MSc informaatikas
2005–2008	Tartu Ülikool, BSc informaatikas
2002–2005	Kuressaare Gümnaasium, keskharidus
1993–2002	Kuressaare Gümnaasium, põhiharidus

Teenistuskäik

2013–	Cybernetica AS, nooremteadur
2011–2013	Cybernetica AS, programmeerija
2010–2011	Tarkvara Tehnoloogia Arenduskeskus OÜ, tarkvaraarendaja

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS