NTNU – Trondheim
Norwegian University of
Science and Technology

A Secure Multi-Party Computation Protocol Suite Inspired by Shamir's Secret Sharing Scheme

Tiina Turban

Master in Security and Mobile Computing

Submission date:	May 2014
Supervisor:	Stig Frode Mjølsnes, ITEM
Co-supervisor:	Dan Bogdanov, Cybernetica, Estonia
	Sven Laur, University of Tartu

Norwegian University of Science and Technology
Department of Telematics

NTNU – Trondheim
Norwegian University of
Science and Technology

A Secure Multi-Party Computation Protocol Suite Inspired by Shamir's Secret Sharing Scheme

Tiina Turban

Submission date:	May 2014
Responsible professor:	Stig Frede Mjølsnes, NTNU
Supervisor:	Sven Laur, UT
Instructor:	Dan Bogdanov, Cybernetica AS

Norwegian University of Science and Technology
Department of Telematics

Abstract

The world today is full of secrets. Sometimes, we would like to know something about them without revealing the secrets themselves. For example, whether I have more money than my friend or whether two satellites would collide without publishing their moving trajectories. Secure multi-party computation allows us to jointly compute some functions while keeping the privacy of our inputs. Sharemind is a practical framework for performing secure multi-party computations. In this work, we added a protocol suite to Sharemind. This protocol suite was inspired by Shamir's secret sharing scheme, which describes a way to divide a secret into pieces. We describe algorithms for addition, multiplication, equality-testing and less-than comparison. We also give correctness and security proofs for the protocols. The resulting implementations were compared to an existing protocol suite inspired by additive secret sharing. The initial complexities and benchmarking results are promising, but there is room for improvement.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

n	number of computing parties.
k	threshold in Shamir's secret sharing scheme.
F	finite field we are working in.
p	prime indicating the finite field Z_p we are working in.
$[\![a]\!]$	secret-shared value a.
$[\![a]\!]_i$	share of secret-shared a, that party $\mathcal{CP}_i$ sees publicly.
$\mathbf{a}$	vector of booleans containing bits of a.
$[\![\mathbf{a}]\!]$	vector of secret-shared values (containing secret-shared bits of a).
$\mathbf{a}_i$	i-th bit of a.
$[\![\mathbf{a}_i]\!]$	secret-shared i-th value from the vector $\mathbf{a}$ (secret-shared bit i of a).
ℓ	length in bits for current datatype in algorithms.
b	length in bits for current datatype in complexity analysis.

List of Acronyms

$\mathcal{CP}$ Computing Party.

$\mathcal{IP}$ Computing Party.

$\mathcal{RP}$ Computing Party.

BLT Bitwise Less-Than.

CPU Central Processing Unit.

EQ Equality.

FairplayMP Fairplay Multy-Party.

GB Gigabyte.

GHz Gigahertz.

GNU LGPL GNU Lesser General Public License.

IP Internet Protocol.

LSB Least Significant Bit.

LT Less-Than.

MEVAL Multi-party EVALuator.

NTNU Norwegian University of Science and Technology.

PD Protection Domain.

PDK Protection Domain Kind.

RAM	Random-Access Memory.
SEPIA	Security through Private Information Aggregation.
SFDL	Secure Function Definition Language.
SMC	Secure Multy-Party Computation.
TASTY	Tool for Automating Secure Two-partY computations.
TASTYL	TASTY input Language.
UT	University of Tartu.
VIFF	Virtual Ideal Functionality Framework.

Chapter 1 Introduction

1.1 Motivation

In today's world, we have a huge amount of information. That data could be used to figure out trends which could, for example, allow us to make wiser business decisions. If we would live in a world without secrets and where everyone trusts each other, then we could simply publish all the information and analyse it. In the real world, however, there are many things, that people consider private, such as their medical or financial details. Companies have business secrets, which they do not want to reveal either. Therefore, it would be great, if there would be a way to analyse data without compromising anyone's privacy. The latter is exactly what secure multi-party computation (SMC) allows us to do.

One of the frameworks that can be used in practice for secure multi-party computation is Sharemind [Bog13]. There are different cryptographic primitives, that secure multi-party computation can rely on, such as homomorphic encryption, additive or Shamir's secret sharing. So far, all the protocol suites implemented on Sharemind fix the number of participants in the computation. Also, if any of the participants would disappear, then we cannot access the results. Protocols using Shamir's secret-sharing, which have not been implemented on Sharemind so far, would provide more flexibility. In theory we could allow more corrupted parties.

Most of the research that exists about SMC using Shamir's scheme only focuses on unsigned integers, especially when dealing with equality testing or comparison operators. We on the other hand are interested in a more universal framework, that allows protocols to be used on both unsigned and signed integers as well as boolean data types.

1.2 Contribution of the author

The goal of this work is to create a new protocol suite for Sharemind and compare it to the existing additive three-party protocols. The implementation shall use Shamir's secret sharing scheme. The implemented protocol suite consists of classification, declassification, resharing, addition, subtraction, multiplication, equality testing and less-than comparison with the necessary sub-protocols. There were various alternatives to be considered for each algorithm. The author developed an experimental implementation on Sharemind. In addition, the author wrote down the algorithms with correctness and security proofs. Finally, the author benchmarked the performance between the new and an existing comparable protocol suite.

1.3 Outline

The List of Symbols on page xv defines the notation used in this thesis. Section 2 gives an overview of the background information. This includes explaining secret-sharing and, more in depth, the Shamir's secret sharing scheme. We describe what secure multi-party computation is and how Sharemind works. There is also a subsection that talks about other SMC frameworks.

Section 3 focuses on the details of implementing a protocol suite on Sharemind. This section shows how to use the result of this work. We also describe how our different data types are represented and the concepts used in proofs.

Section 4 shows how our private information can be taken into pieces and divided among computing parties. The declassification subsection, on the other hand, shows how the computed result, that is still in secret-shared form, can be reconstructed to publish the value.

The arithmetic protocols are given in Section 5 and algorithms for comparison operations in Section 6. The latter includes various sub-protocols, such as least significant bit, that were needed for equality testing or less-than comparison.

Section 7 sums up the complexities and brings out the benchmarking results with comparison to the additive three-party protocol suite. The final Section concludes this thesis and provides ideas for further work.

Chapter 2
Preliminaries

2.1 Secret sharing

Secret sharing [Sha79, Bla79] is a technique for protecting confidential data. The secret is divided into parts — shares. These shares will then be distributed among a number of parties. In order to reconstruct the secret, a certain predefined set of shares must be combined. For example, unique shares are divided to n participants, but any k of them together can retrieve the original secret. This structure is also known as k-out-of-n threshold scheme. Gaining access to less than that threshold k of distinct shares shall give no information about the secret.

Definition 1. Let s be a secret value and $[\![s]\!]_1, [\![s]\!]_2, ..., [\![s]\!]_n$ shares. We have a k-out-of-n secret sharing scheme, if the following conditions hold [Sha79]:

Correctness: knowledge of any k or more shares of s makes the secret easily computable;

Privacy: knowledge of any $k-1$ or fewer shares of s leaves the secret completely undetermined (in the sense that all its possible values are equally likely).

Additive secret sharing scheme is a form of secret sharing. It is a scheme, where one needs to know all the shares to discover the original value, i.e. an n-out-of-n threshold scheme. The algorithm divides shares by first uniformly choosing $n-1$ values $[\![s]\!]_1, [\![s]\!]_1, ..., [\![s]\!]_{n-1}$ and then calculating $[\![s]\!]_n = s - [\![s]\!]_1 - ... - [\![s]\!]_{n-1}$. The secret s can be reconstructed by adding all the shares together $s = [\![s]\!]_1 + ... + [\![s]\!]_n$, but knowing $n-1$ or less shares gives a malicious entity (adversary) no information about s.

Shamir's secret sharing scheme [Sha79] is a form of secret sharing, which uses the idea that k points are needed to uniquely define a polynomial of degree $k-1$. With this scheme, a threshold k can be chosen, which defines the number of shares needed for reconstructing the secret, i.e. it is a k-out-of-n scheme.

Shares in Shamir's secret sharing scheme have two values - an index and the evaluation of the randomly generated polynomial on that index – $(i, f(i))$. The indices have to be unique to each party and we cannot use zero as that would reveal the secret $f(0) = s$. To create the shares, we choose n random points on the polynomial $(x,\ f(x))$. These points we then distribute to the parties. Coefficients will be chosen randomly and the free member is equal to the secret. To illustrate how it works, we can look at Figure 2.1. In reality, Shamir's scheme uses polynomials over a finite field and not a two dimensional plane. In this example, we use a 2-out-of-4 threshold scheme, where $[\![s]\!]_1, ..., [\![s]\!]_4$ represent the shares for a secret value s. The constant c_1 is random and hence $f(x) = s + c_1x$ is a random 1-degree polynomial.

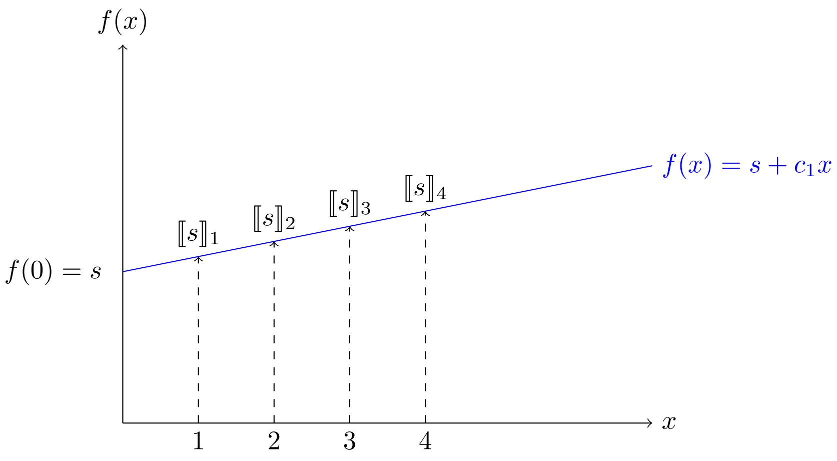

Figure 2.1: Classifying a secret value with Shamir's secret sharing scheme

Figure 2.2 illustrates the situation, where we know less than k shares. If we know only one point for a 1-degree polynomial, we can draw an infinite number of lines covering all the possible values ($s, s', s'', ...$ on Figure 2.2) and there is no way to know which one is the right one.

However, if we know at least k shares, it is possible to reconstruct the originally created polynomial $f(x)$. Figure 2.3 illustrates, that there is only one straight line, we can draw through two points on a two dimensional plane. We can use Lagrange interpolation [WR67]. Let $t_1, ..., t_k$ be the indices for any unique k shares $(t_i, f(t_i))$. According to the Lagrange interpolation formula, we can evaluate a polynomial at

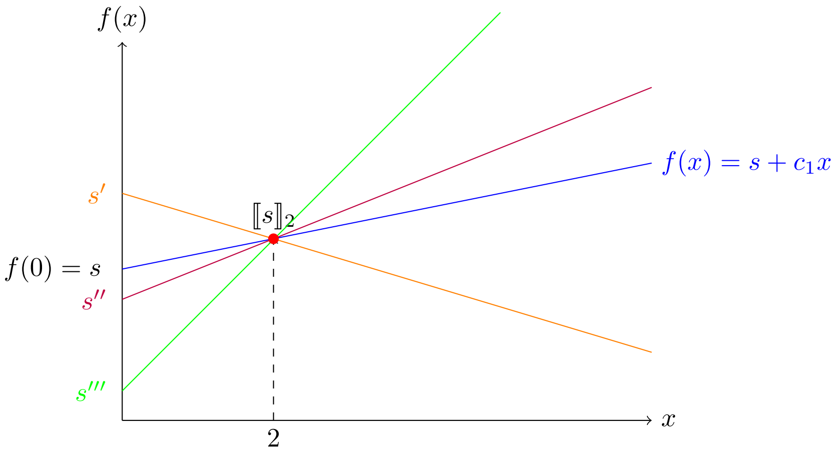

Figure 2.2: A failed attempt to reconstruct a secret knowing $k - 1$ shares.

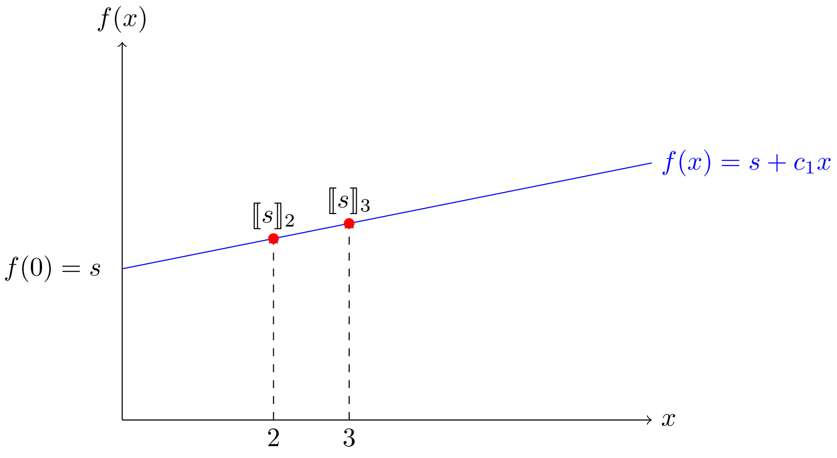

Figure 2.3: Reconstructing the secret with Shamir's secret sharing scheme

any given value by calculating

$$f(x) = \sum_{i=1}^{k} f(t_i) b_i(x),$$

where $b_i(x)$ is the Lagrange basis function

$$b_i(x) = \prod_{\substack{j=1 \\ i \neq j}}^{k} \frac{x - t_j}{t_i - t_j}.$$

To find the secret we are only interested in the value of the polynomial in position zero, i.e. $f(0)$. Let b_i denote the Lagrange basis function $b_i(0)$. Then,

$$b_i = b_i(0) = \prod_{\substack{j=1 \\ i \neq j}}^{k} \frac{0 - t_j}{t_i - t_j} = \prod_{\substack{j=1 \\ i \neq j}}^{k} \frac{-t_j}{t_i - t_j}. \tag{2.1}$$

Hence, the secret can be found with

$$s = f(0) = \sum_{i=1}^{k} f(t_i) b_i. \tag{2.2}$$

Note that the Lagrange basis function does not depend on the polynomial, which means that b_i's can be pre-calculated.

Alternatives. Homomorphic encryption [Gen09] can be used instead of secret sharing, however the latter has been shown to be less efficient in practice [KBdH09]. Another alternative is Yao's garbled circuit construction [Yao86].

2.2 Secure multi-party computation based on secret sharing

With secret sharing we can protect our data, but often we would still like to process the protected data without compromising privacy. That is exactly what secure multi-party computation allows us to do – compute various functions without giving away any information about their own shares. Some real-life use cases include benchmarking, where several companies want to compare themselves to each other [BTW12], various forms of auctions and private biddings [BCD+09].

Secure Multy-Party Computation (SMC) was initially introduced in 1982 by Yao [Yao82]. In his paper, Yao brings the example of two millionaires wanting to know who is richer without revealing their wealth. More generally, we have any number of input parties $\mathcal{IP}_1$, $\mathcal{IP}_2$, ..., $\mathcal{IP}_m$, computing parties $\mathcal{CP}_1$, $\mathcal{CP}_2$, ..., $\mathcal{CP}_n$ and result parties $\mathcal{RP}_1$, $\mathcal{RP}_2$, ..., $\mathcal{RP}_r$. The computing parties wish to compute $f([\![x]\!]_1, [\![x]\!]_2, ...[\![x]\!]_n) = ([\![y]\!]_1, [\![y]\!]_2, ...[\![y]\!]_n)$. Initially, each party $\mathcal{CP}_i$ knows $[\![x]\!]_i$, but no other $[\![x]\!]_j$, $i \neq j$. After jointly computing the function f, each party learns their output $[\![y]\!]_i$ and nothing else. Some, or all of the output values can be equal. For example, in the millionaires' problem, we wish to compute

$$f([\![x]\!]_1, [\![x]\!]_2) = \begin{cases} 1 & \text{if } [\![x]\!]_1 < [\![x]\!]_2, \\ 0 & \text{otherwise} \end{cases}$$

where both parties get the same result ($[\![y]\!]_1 = [\![y]\!]_2$).

There are different security requirements that may be needed for different applications. Cheating in the context of secret sharing and secure multi-party computation can be seen as having an adversary, who may corrupt some subset of computing parties [CD05]. Corruption can either be passive or active. In the first case the adversary can see all the data the corrupted parties have. Active adversary can, in addition to seeing everything, also manipulate the messages sent or even stop sending anything altogether. Adversaries may additionally be divided into static and adaptive. If there is a constant set of corrupted players over the course of running the protocol then we are dealing with a static adversary. Adaptive adversaries, on the other hand, can choose at any point in time to corrupt a different set of players. In this thesis, we are considering protocols with static, passive adversaries. This is also known as the honest-but-curious security model. We can have at most $k-1$ corrupt parties, otherwise the adversary has enough shares to reconstruct the secret.

2.3 Sharemind

2.3.1 The Sharemind secure computing framework

Sharemind is a framework for building data processing applications that use secure multi-party computation. It is designed with the intention to be efficient enough for practical applications, but at the same time to be usable by non-cryptographers [BLW08, Bog13]. The practical part of this thesis is implemented on the Sharemind SMC framework and more specifically on version 3 of Sharemind.

An example system setting is illustrated on Figure 2.4. There are three different kinds of parties: input parties ($\mathcal{IP}$), computing parties ($\mathcal{CP}$) and result parties ($\mathcal{RP}$). Participants are not restricted to belonging to only one of these groups, but can also be all of the above or just an input party, who wants to learn results. There can be any number of input and result parties, but the number of computing parties might be restricted by the protocols used. $\mathcal{IP}$s use secret sharing or other techniques to distribute their confidential data between the $\mathcal{CP}$s. $\mathcal{RP}$s make queries and initiate computations, that are performed by $\mathcal{CP}$s on the shared data. In the end, $\mathcal{RP}$s get the public computation results without anyone seeing the original confidential data.

We could, for example, run 2-out-of-4 Shamir's secret sharing scheme on Sharemind 3. Figure 2.4 shows an $\mathcal{IP}$ classifying a secret s_1 by giving each $\mathcal{CP}$ one share.

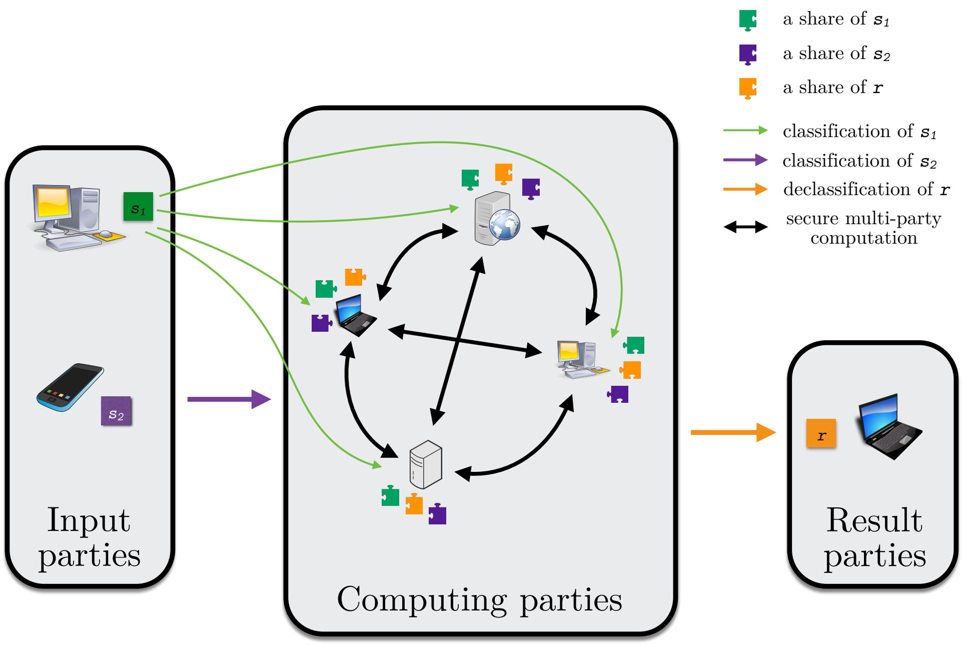

Figure 2.4: Sharemind 3 deployment model

Another $\mathcal{IP}$ can secret-share another secret s_2. Via secure multi-party computation we can then calculate $s_1 < s_2$, where each $\mathcal{CP}$ has only one share of the result. To reconstruct the secret-shared result r, at least two $\mathcal{CP}$s must send their shares to the $\mathcal{RP}$ (because we were using 2-out-of-4 scheme).

Computing parties perform computations by executing algorithms, which consist of addition, multiplication or other operations. These operations are evaluated by running protocols, that are described using some cryptographic primitives. A Protection Domain Kind (PDK) defines a set of data representations for storing and protocols for computing on protected data [BLR13]. A PDK can be designed for any secure computation techniques with different security guarantees and different PDKs can support different operations. Each PDK can have several Protection Domains (PD), which are concrete initialisations of a PDK. For example, in this work we will create a PDK that uses the k-out-of-n Shamir's secret sharing scheme. For that PDK, we can define a PD with concrete n and k values, e.g. a 2-out-of-4 scheme. The first PDK that was designed for Sharemind uses additive secret sharing with three $\mathcal{CP}$s and it is secure in the semi-honest model [BLW08, Bog13]. In 2013, another PDK using additive secret sharing was described for two computing parties that offers security against active adversaries [Pul13]. Another example of a PD is fully homomorphic encryption scheme with addition and multiplication protocols, and a pair of keys. We can have another PD of the same PDK that differs only in protection keys, i.e. its configuration.

Sharemind has been used for various practical applications. A recent example demonstrates, how secure multi-party computation can be used for calculating the probability of a collision between two satellites [KW13]. Countries do not want to reveal exact information about their satellites nor do they want to lose them. But a collision in 2009 demonstrates, that knowing only approximate data about the orbits is not enough. It was shown, that SMC can be used as a possible solution. Another example comes from Estonian Association of Information Technology and Telecommunications, who wished to calculate benchmarking results based on their economic indicators [Tal11, BTW12]. Their initial solution had some security-related shortcomings and a new solution with stronger privacy guarantees using Sharemind was proposed. This was the first time where SMC computation on real data was done over the internet with geographically apart computing nodes. The bioinformatics field offers a third example, where secure multi-party computation could be used to protect the privacy of individuals participating in a study [KBLV13].

2.3.2 Protection domain deployment configuration

To make the deployment easily configurable, we have separate files, that define parameters for each protection domain and each computing party [AS11]. They contain addresses and encryption keys of other $\mathcal{CP}$s. The configuration files can also have constants, such as fragment size for controlling parallelism. These constants can then be used in the protocols described in that protection domain kind. There can be any number of computing parties, of which some can be non-computing nodes for certain PDs. The deployment configuration is not limited to having only one protection domain, but we can describe and use PDs in parallel. For example, see Table 2.1, where we have three protection domains defined on our four $\mathcal{CP}$s. Computing nodes are denoted with a star (*) in the table.

Protection domain	$\mathcal{CP}_1$	$\mathcal{CP}_2$	$\mathcal{CP}_3$	$\mathcal{CP}_4$
Additive 3-party	*	*	*	
FHE		*		
Shamir 2-out-of-4	*	*	*	*

Table 2.1: Multiple protection domains deployment configuration

2.3.3 SecreC 2

SecreC [Jag10] is a privacy-aware programming language inspired by C. Its second version SecreC 2 [BLR13] is used in the Sharemind 3 SMC framework. It is used to describe algorithms that run on $\mathcal{CP}$s to calculate results for $\mathcal{RP}$s. The language is designed to be easy to use and the programmer can just call the PDK protocols and operations as predefined functions, i.e. he/she does not need to understand the underlying cryptographic primitives. To make the developers life easier, there is an integrated development environment for the SecreC programming language (SecreCIDE) [Reb10].

SecreC 2 is strongly typed, where the type has a data type and a PD. There is a predefined PD for public types and it can be omitted when defining public variables, i.e. `int x;` would define a public integer. It is a polymorphic language that allows to write code not specific to a certain PDK. Obviously, that PDK must define the protocols used in the code. Being domain-polymorphic allows for an easy integration of new PDs or re-usage of code for different deployment scenarios or common functionality. The latter is also the reason, why it makes sense to have a standard library for SecreC 2. The standard library includes functions, such as minimum, maximum and absolute value. Additionally, if for a specific PDK, there is a more efficient version, it is possible to implement a special version aside the general function.

```
import additive3pp;
import shamirnpp;
domain a3pp additive3pp;
domain s2of4 shamirnpp;

template <domain D, type T>
D T abs (D T x) {
  return x < 0 ? -x : x;
}

void main {
  a3pp uint x = 5;
  a3pp uint ax = abs(x);
  assert(declassify(ax) == (5 :: uint));

  s2of4 int y = -5;
  s2of4 int ay = abs(y);
  assert(declassify(ax) == (5 :: int));
}
```

Listing 2.1: SecreC 2 example – Absolute value

In order to use protection domains in SecreC 2, we need to define them. We have additive secret sharing for three parties against passive adversary implemented for Sharemind, so the module `additive3pp` can simply be defined or imported. The latter provides us additionally the possibility to use `additive3pp`'s standard library. After which we can define a protection domain `a3pp` that can be used in a domain-polymorphic function. Listing 2.1 gives an example of SecreC 2 code with two PDs.

The PDK `shamirnpp` will be created with this thesis and PD *s2of4* can be defined for it. The function *abs* can be used with any domain and any type as long as there is less-than and additive inverse defined on $D\ T$ types. The *main* function calls *abs* on both PDs on different types and then checks that the value was as expected.

2.4 Other SMC frameworks

Even though the theory of secure computations has been around since the eighties [Yao82], the first practical implementations were introduced after the millennium. Several frameworks, such as Fairplay [MNPS04], SEPIA [BSMD10], VIFF [Gei10], TASTY [HKS+10], VMCrypt [Mal11], MEVAL [CMF+14] and PICCO [ZSB13] have been developed since.

Fairplay[1] was the first practical implementation of SMC and it was introduced in 2004. The initial version used Yao's garbled circuits [Yao86] and supported secure communication between two parties. In 2006 Ben-David, Nisan and Pinkas created an extension of the system called FairplayMP, for Fairplay Multi-Party. This version uses Yao circuits and ($\lfloor \frac{n}{2} \rfloor + 1$)-out-of-$n$ secret sharing. They have their own high-level programming language called Secure Function Definition Language (SFDL), in which users can write code, that will then be compiled into a low-level representation as a Boolean circuit. To run secure multi-party computation, users must also write a configuration file with IP addresses and other settings.

FairplayMP is implemented in Java, focusing on performance in terms of message sizes and the number of communication rounds. To check whether the system could be used for real life problems, the authors experimented with protocols for voting and computing auctions. More specifically, they ran a second-price auction [Vic61] (winner pays the amount of second-highest bid) between bidders, where everyone learns the second-highest bid, but only the seller learns the identity of the winner. In total, there were five computing parties, for each a computer with two Intel Xeon 3 GHz CPU processors and 4 GB of RAM was used. Running the experiment for 8-bit bids took about 8 seconds [BDNP08].

VIFF[2] was originally developed in the Secure Computing Economy and Trust (SCET) and the Secure Information Management And Processing (SIMAP)[3] projects [BDJ+06]. The technology developed during those projects was deployed to run the first large-scale SMC in 2008 [BCD+09]. The practical experiment was ran

[1]Fairplay – http://www.cs.huji.ac.il/project/Fairplay/
[2]VIFF – http://viff.dk
[3]SIMAP Project – http://www.alexandra.dk/uk/projects/pages/simap.aspx

with Danish farmers trading sugar beet contracts using a secure double auction. The Virtual Ideal Functionality Framework (VIFF) is implemented in Python and is Free Software, licensed under the GNU LGPL[4]. It uses Shamir and pseudo-random secret sharing [CDI05]. Various protocols have been implemented of VIFF. In addition to passive, it is also possible to write protocols that are secure against active and adaptive adversaries. For example, they implemented multiplication that is secure against malicious adversary and, for 7 computing parties, it took 2.7 seconds to prepare 1000 multiplications, but only 2 seconds to execute all of them [Gei10].

SEPIA[5], which is short for Security through Private Information Aggregation is a Java library for SMC. It is also Free Software, licensed under the GNU LGPL. SEPIA uses Shamir's secret sharing and, similarly to Sharemind's `additive3pp` protection domain, it is secure against static passive adversaries. In 2010, SEPIA outperformed VIFF and FairplayMP for running multiplication and comparison operations in parallel. Compared with Sharemind version 2 however, performance was similar [BSMD10].

TASTY[6] is a Tool for Automating Secure Two-partY computations. This tool uses homomorphic encryption or garbled circuits or their combinations to automatically generate efficient protocols from their high-level description. They have their own specification language called TASTY input language (TASTYL), which is based on Python, as TASTY itself is implemented in Python. TASTY's Runtime Environment also provides the possibility to automatically analyse, run, test, and benchmark the two-party secure function evaluation protocol. Comparing the performance to original Fairplay, which also uses Yao's garbled circuits construction, TASTY requires less memory, communication and online time, though the setup time is slower. Henecka, Kögl, Sadeghi, Schneider, and Wehrenberg assume that this is due to their implementation language choices (Python versus Java) [HKS+10].

VMCrypt is a software library created with a goal to be modular and scalable. It is implemented in Java and uses Yao's garbled circuits. They noticed that in order to make the system scalable, they need to look at memory consumption, as holding large circuits in memory would take too much RAM. Hence, VMCrypt takes a streaming approach to generating circuits. It streams the circuit gate by gate, i.e. when a part of the circuit is ready, it will be passed to the evaluator and the computation process can already begin. This allowed Malka to run performance tests on circuits with

[4]GNU LGPL – https://www.gnu.org/licenses/lgpl.html
[5]SEPIA – http://sepia.ee.ethz.ch
[6]TASTY – https://code.google.com/p/tastyproject/

hundreds of millions of gates [Mal11].

PICCO [ZSB13] is a general-purpose compiler for private distributed computation. Input for the compiler is a program, written in a C language extension, that provides a way to annotate private data. The output will be its secure distributed implementation in C. The resulting code can then be compiled with a native C compiler and executed by a number of computation nodes. Zhang, Steele and Blanton also concentrated on performance and making the secure computation scalable. To do that, they implemented multiple types of parallelism, e.g. over loops and arrays. Internally, PICCO uses Shamir's secret sharing.

MEVAL [CMF$^+$14], which is short for Multi-party EVALuator is a SMC system. It uses Shamir's secret sharing scheme and provides security against passive adversaries. For better performance, they use asynchronous processing and a Mersenne prime field to get optimised field operations. At the Applied Multi-Party Computation workshop at Microsoft Research, Hamada gave the following performance results in his presentation [Ham14]. 8.7 multiplications of 61-bit integers can be done per second and sorting 1 million 20-bit items takes 6.9 seconds. MEVAL uses R[7] with an add-on as a front-end client application.

[7] R - http://www.r-project.org

Chapter 3

A protection domain kind based on Shamir's secret sharing

3.1 Protection domain setup

The goal of this work is to provide a way to use Shamir's secret sharing on Sharemind. We will create a new protection domain kind `shamirnpp`, that can be used with various number of $\mathcal{CP}$s. Previously, the PDKs have defined the number of computing parties, e.g. `additive3pp` has three $\mathcal{CP}$s. The protection domain specifies n and k values, for k-out-of-n Shamir's secret sharing scheme. The threshold is needed for the classification protocol, as we need to know what degree random polynomial to create. To achieve that our PD configuration files also contain the constant k, that is used in the PDK protocols.

3.2 Data types supported by the protection domain kind

3.2.1 Unsigned integers

We have different unsigned types, such as `uint8`, `uint16`, `uint32` and `uint64`. We could only have one type, but there are trade-offs here – it is cheaper for network communication and memory to calculate using `uint8`, but 256 values is often not enough.

Shamir's secret sharing is done on a field, so we shall work in a finite field. That means there should be prime number possible values. Our default data types in the computer for integers however have powers of two values.

To make life easier we just use the largest prime value in the finite field, that fits in n bits as the maximum value, that an n-bit integer type can have. Table 3.1 shows, for each unsigned type, its size (the number of values a certain type can hold), the largest prime, i.e. how many values our type will be able to hold and the last column contains the number of lost values. If one goes over the maximum value then overflow happens, e.g. $200 + 55 = 5$.

Type	sizeof(`uintX`)	largest prime (p)	difference (d)
`uint8`	256	251	5
`uint16`	65536	65521	15
`uint32`	4294967296	4294967291	5
`uint64`	18446744073709551616	18446744073709551557	59

Table 3.1: Unsigned integers for Shamir secret sharing on Sharemind

We know that our `uintX` is not a real X-bit unsigned integer, but our goal is to achieve comparability with other PDKs. Aside from the fact, that last few values cannot be used, it does not influence the developer's life.

3.2.2 Implementing calculations modulo p

In order to more comfortably do operations in our finite field, i.e. modulo a prime p, we created our own types. Internally they still use default types supported by the processor and have 2^n values, which means that in case of an overflow, the result would be wrong. There are multiple solutions available here, for example

1. performing the operations in a bigger type and using modulo p in the end,
2. making corrections, i.e. adding or subtracting the difference from the result when needed.

Upcasting the type and performing operations there might not be so efficient if we run out of bigger default types, for example for `uint64`.

Addition is done by making corrections, see Algorithm 1. The result of addition can end up in three different regions: $[0, p)$, $[p, 2^n)$ or $[2^n, 2p - 2)$. The last one result in an overflow in the native type and we need to add d to correct it for our `uintX`. The middle region, on the other hand, should have been overflown.

Subtraction is done by making corrections, see Algorithm 2. The result of subtraction can end up in two different regions: $(-p, 0)$ or $[0, p)$. In the first case, we subtract d to correct the overflow.

Algorithm 1: Implementing addition in Z_p on native types

```
Input: Prime $p$, values $a, b \in Z_p$, difference $d$
Result: $c \in Z_p$
1 $c = a + b$
2 if $c < a$ then
3     $c = c + d$
4 else if $c \geq p$ then
5     $c = c - p$
```

Algorithm 2: Implementing subtraction in Z_p on native types

```
Input: Prime $p$, values $a, b \in Z_p$, difference $d$
Result: $c \in Z_p$
1 $c = a - b$
2 if $c > a$ then
3     $c = c - d$
```

Multiplication is simply done by converting to a larger type and applying the modulus after multiplication.

Multiplicative inverse is found using the Extended Euclidean algorithm.

3.2.3 Signed integers

When thinking about signed integers for secret sharing, we can look at a somewhat similar problem of how negative numbers are represented in computer hardware. There are four best-known methods: sign and magnitude notation, one's complement, two's complement and excess-K representation. Even though two's complement is most widely used, there are advantages and disadvantages to each representation [Flo63]. In this section, we will be considering three different ideas for signed integers notation, each having its own benefits and drawbacks.

Sign and magnitude. When keeping the sign separately from magnitude we can use the first bit, but we can also just use a separate boolean value, which might make things easier later. For example, we do not need to extract the most significant bit which, in secret sharing, is not that trivial and can use the separate boolean value instead. This representation makes it easy to perform multiplication, but for addition and subtraction, getting the sign right is not so trivial.

Modified two's complement. Another idea is to split the value range into positive and negative parts. We got the idea from two's complement notation for signed integers in hardware. Since we do not use the full range of values in unsigned integers, e.g. `uint8` maximum value is 250 (11111010_2), we need to modify the notation by adding or subtracting the difference d. Otherwise, we do not have small negative numbers as they would result in an overflow, e.g. $-1 = 11111111_2 = 00000101_2 = 5$. Hence, to convert the negative signed integer i to the internal unsigned representation u we find the two's complement and subtract d. Vice versa, i.e. from u to i, we add d to the negative value found by taking the two's complement of u.

Table 3.2 shows the mapping between unsigned and signed integers. Additionally, the corresponding values and the bitwise representation are given for 8-bit integer types ($p = 251$).

unsigned	**signed**	`uint8`	`int8`	**Binary(`uint8`/`int8`)**
0	0	0	0	00000000
1	1	1	1	00000001
...	...	...	...	...
$\lfloor \frac{p}{2} \rfloor - 1$	$\lfloor \frac{p}{2} \rfloor - 1$	124	124	01111100
$\lfloor \frac{p}{2} \rfloor$	$\lfloor \frac{p}{2} \rfloor$	125	125	01111101
$\lfloor \frac{p}{2} \rfloor + 1$	$\lfloor \frac{p}{2} \rfloor + 1$	126	126	01111110
...	...	...	...	...
$\lfloor \frac{p}{2} \rfloor + \lfloor \frac{d}{2} \rfloor$	$\lfloor \frac{p}{2} \rfloor + \lfloor \frac{d}{2} \rfloor$	127	127	01111111
$\lfloor \frac{p}{2} \rfloor + \lfloor \frac{d}{2} \rfloor + 1$	$\lfloor \frac{d}{2} \rfloor - \lfloor \frac{p}{2} \rfloor$	128	-123	10000000
$\lfloor \frac{p}{2} \rfloor + \lfloor \frac{d}{2} \rfloor + 2$	$1 + \lfloor \frac{d}{2} \rfloor - \lfloor \frac{p}{2} \rfloor$	129	-122	10000001
...	...	...	...	...
$p - 2$	-2	249	-2	11111001
$p - 1$	-1	250	-1	11111010

Table 3.2: Signed integers based on the most significant bit

Centered around zero. The third idea also splits the value range into positive and negative parts. Algesheimer, Camenish and Shoup [ACS02] described how it can be done by keeping the values centered around zero. Hence, to convert the signed integer i to the internal unsigned representation, we simply add p, if $i < 0$. Getting the signed value back would mean subtracting p from $u \in Z_p$ if $u > \lfloor \frac{p}{2} \rfloor$. Notice that we cannot only look at the most significant bit to determine if the value is negative, since our unsigned types do not use all the values compared to the native types in computer, see Section 3.2.1. This might make it difficult to use bits in various algorithms. However, we can use the comparison $a < \lfloor \frac{p}{2} \rfloor$ to determine if a contains a negative value as an alternative to looking up the most significant bit in two's complement.

Table 3.3 shows the mapping between unsigned and signed integers. Additionally, the corresponding values and the bitwise representation are given for 8-bit integer types ($p = 251$).

unsigned	**signed**	`uint8`	`int8`	**Binary (`uint8`/`int8`)**
0	0	0	0	00000000
1	1	1	1	00000001
...	...	...	...	...
$\lfloor \frac{p}{2} \rfloor - 1$	$\lfloor \frac{p}{2} \rfloor - 1$	124	124	01111100
$\lfloor \frac{p}{2} \rfloor$	$\lfloor \frac{p}{2} \rfloor$	125	125	01111101
$\lfloor \frac{p}{2} \rfloor + 1$	$-\lfloor \frac{p}{2} \rfloor$	126	-125	01111110
$\lfloor \frac{p}{2} \rfloor + 2$	$1 - \lfloor \frac{p}{2} \rfloor$	127	-124	01111111
$\lfloor \frac{p}{2} \rfloor + 3$	$2 - \lfloor \frac{p}{2} \rfloor$	128	-123	10000000
...	...	...	...	...
$p - 2$	-2	249	-2	11111001
$p - 1$	-1	250	-1	11111010

Table 3.3: Signed integers centered around zero

Conclusion. The sign and magnitude separation makes it easy to understand what value is represented, but this does not overcome the increased complexity for addition and subtraction. Hence, the choice is left between our modified two's complement or centering around zero. For both options, the basic arithmetic protocols simply work on the underlying unsigned representations. When we think about comparison operators, and more specifically the less than operation, initially it seems, that we need to perform bitwise operations. In that case, having the most significant bit denote the sign becomes useful. But as it turns out, there is a more efficient way to compute less-thans using a comparison to half of the prime (see Section 6.3). This makes centering around zero a better choice for the protocols implemented in this work.

3.2.4 Booleans

It is tempting to use a finite field Z_2 to represent booleans. However, this would restrict us to only having two unique shares. For example, for $k = 2, n = 3$ we would create a polynomial $f(x) = s + cx$ (see Section 4.1 for information on how values are classified). Now being in Z_2 would mean that $p = 2$ and $f(x) = s + cx \bmod 2$. Hence the odd number shares are equal

$$\begin{aligned} f(1) &= s + c \\ f(2k+1) &= s + (2k+1)c = s + c \bmod 2. \end{aligned}$$

Then, in our example 2-out-of-3 scheme, we cannot reconstruct the secret having $\mathcal{CP}_1$ and $\mathcal{CP}_3$ (see Section 2.1 for information about declassification with Shamir's secret sharing scheme). What is even worse, the even number shares reveal the secret s.

$$f(2) = s + 2c = s \bmod 2$$

Our solution is to just use 8-bit unsigned integers to represent booleans, having $s \in \{0, 1\}$.

3.3 Security model

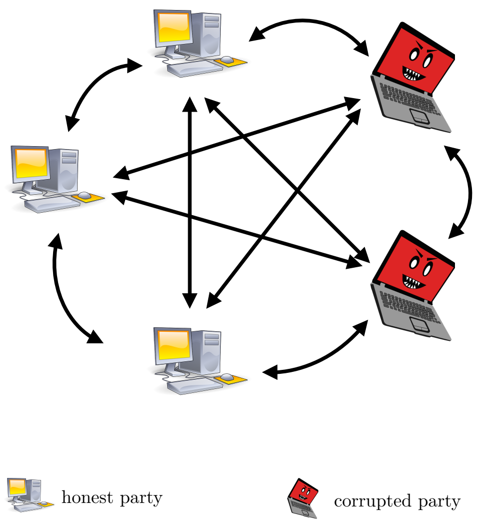

Figure 3.1: Sharemind in the real world setting

To prove the security of our protocols we will use the security proof framework described for `additive3pp` [Bog13]. We have the real world (see Figure 3.1), where $\mathcal{CP}$s exchange messages between each other to calculate some function f. We define an ideal world (see Figure 3.2), where there is a trusted third party, that collects the inputs and calculates f. To prove security, we show, that any real world attack also exists in the ideal world. We do that using perfect simulatability, which ensures that

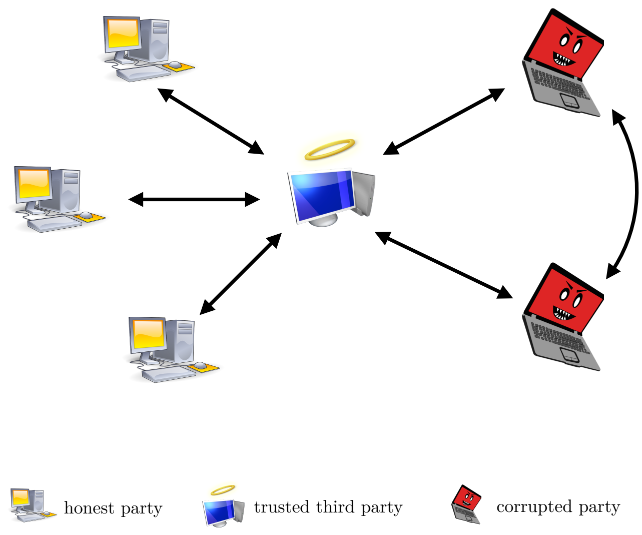

Figure 3.2: Sharemind in the ideal world setting

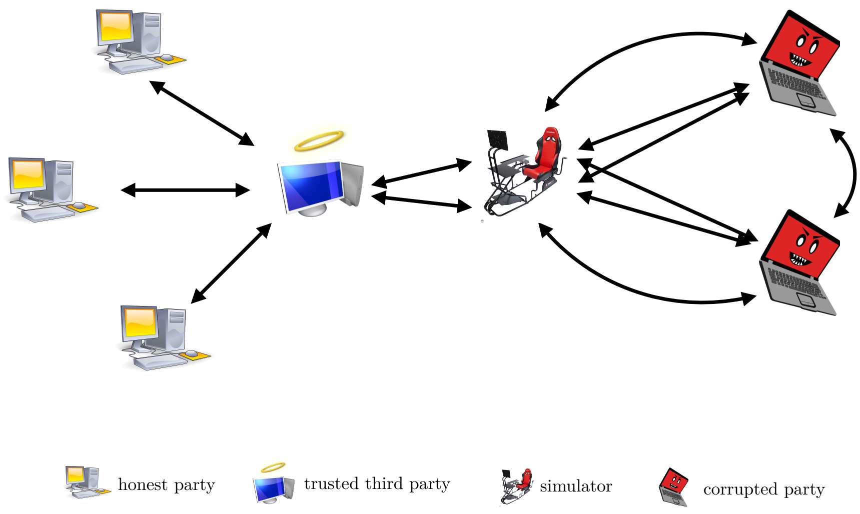

Figure 3.3: Perfect simulation

the adversary cannot distinguish between its views of the protocol in the real and the ideal world. Perfect simulatability guarantees that the adversary does not learn anything except what can be derived from corrupted parties' inputs and outputs. To do that, we construct a simulator (see Figure 3.3), that can simulate our protocol in the real and ideal world indistinguishably. The simulator cannot rewind the algorithm to an earlier state. Notice that this is not the standard definition used in cryptographic proofs for simulatability. A protocol, that consists only of perfectly simulatable sub-protocols and has their outputs used only either as inputs to another sub-protocol or outputs of the main protocol, is itself perfectly simulatable [Bog13, Theorem 4]. But if we re-use the output of a perfectly simulatable protocol it may leak information. More precisely, if output shares depend on input shares, then we cannot achieve better security than perfect simulatability. If they are independant, then we have universal composability. The latter can be achieved by resharing (see Section 4.2) in the end of a perfectly simulatable protocol. In 2014, a more detailed version of the security model was published [BLLP14]. In this thesis, we present security proofs in the model of [Bog13]. This means, that we show, that the protocols are correct and either perfectly simulatable or universally composable.

Chapter 4

Basic protocols

In this and the following sections, we shall use $F = Z_p$ to denote the finite field we are operating on. The letter b shall represent the bit-length of p, i.e. the bit-length of value $s \in F$. In the following algorithms, a share of $[\![s]\!]$ for $\mathcal{CP}_t$ is denoted as $[\![s]\!]_t$. When analysing the complexities of our protocols, we notice that some things can be pre-computed and others depend on the inputs. Therefore, we shall separate the offline and online phase. In both phases, we are mainly interested in two things – the number of times a $\mathcal{CP}$ needs to wait for input (rounds) and bits of data transferred (communication cost). Generally, we prioritise minimising rounds over communication costs [Reb12].

4.1 Classification

As mentioned previously, shares in Shamir's secret sharing scheme consist of two values – input to the polynomial and the corresponding output. We have decided to use the $\mathcal{CP}$'s node number t as the first part of the share and then calculate $f(t)$. This way, we do not need to use network resources to communicate them. More-over, the numbers are guaranteed to be unique per protection domain and there is no node number zero. To classify the secret value s, we must first create a random polynomial with degree $k - 1$, where the free term is the secret value, e.g. $k = 3$, $f(x) = s + 12x - 43x^2$. Algorithm 3 is given for an input party wanting to classify a secret value, but any $\mathcal{CP}_t$ (computing party with node number t) can classify a value by sending $f(x)$ to all other $\mathcal{CP}$s and keeping $f(t)$ as their own share. This is also the case when a participant is an $\mathcal{IP}$ and a $\mathcal{CP}$ at the same time.

For boolean values, the same algorithm is used, we just know that $s \in \{0, 1\}$. For signed integers, the secret s is first converted to an unsigned integer u and then classification protocol is run on u. For centering around zero representation, the conversion can be done by adding p to negative inputs. Because then, assuming correct input range, the positive values are in $\{0, ..., \lfloor \frac{p}{2} \rfloor\}$ and negative values are in

Algorithm 3: Protocol for classifying a secret value `\cmdClassify`(s)

```
Input: Finite field $F$, threshold $k$, $\mathcal{IP}$ has an unsigned integer secret $s \in F$
Result: All $\mathcal{CP}$s have a share of $[\![s]\!]$.
1 $\mathcal{IP}$ uniformly chooses $c_1, \ldots, c_{k-1} \overset{u}{\leftarrow} F$
2 $\mathcal{IP}$ constructs the polynomial $f(x) = s + c_1x + c_2x^2 + \cdots + c_{k-1}x^{k-1}$
3 $\mathcal{IP}$ sends $f(x)$ to $\mathcal{CP}_x$
4 Each $\mathcal{CP}_x$ receives $f(x)$
```

$\{-\lfloor \frac{p}{2} \rfloor + p, ..., -1 + p\} = \{-\frac{p-1}{2} + p, ..., p-1\} = \{\lceil \frac{p}{2} \rceil, ..., p-1\}$, see Algorithm 4.

Algorithm 4: Protocol for classifying a signed integer

```
Input: Prime $p$, $\mathcal{IP}$ has a signed integer secret $s \in \{-\lfloor \frac{p}{2} \rfloor, ..., \lfloor \frac{p}{2} \rfloor\}$
Result: All $\mathcal{CP}$s have a share of $[\![s]\!]$.
1 $\mathcal{IP}$ calculates:
2 if $s \geq 0$ then
3     $u = s$
4 else
5     $u = s + p$
6 $\mathcal{IP}$ runs Classify$(u)$
```

If the input party does not spread the polynomial constants and we assume that there is a secure connection from $\mathcal{IP}$ to each $\mathcal{CP}$, then the protocol is secure. Complexity for both unsigned or signed integers requires one round and communication of one unsigned integer to each $\mathcal{CP}$, i.e. nb bits of information. Remember that, for booleans, $b = 8$.

4.2 Resharing

The simplest way for refreshing a secret is by adding a secret-shared zero [NN05].

Secret-sharing a zero. One of the $\mathcal{CP}$'s could use the classify protocol to secret-share zero, but the entity, who does the sharing, would know every $\mathcal{CP}$'s share. There is, however a possibility, using more communication, to secret-share zero without anyone knowing other $\mathcal{CP}$'s shares. To do that, every $\mathcal{CP}_t$ first classifies zero, which means that each $\mathcal{CP}_t$ creates a polynomial g_t

$$\begin{aligned}
g_1(x) &= 0 + c_{1_1}x + c_{1_2}x^2 + \cdots + c_{1_{k-1}}x^{k-1} \\
g_2(x) &= 0 + c_{2_1}x + c_{2_2}x^2 + \cdots + c_{2_{k-1}}x^{k-1} \\
&\dots \\
g_n(x) &= 0 + c_{n_1}x + c_{n_2}x^2 + \cdots + c_{n_{k-1}}x^{k-1}.
\end{aligned}$$

Secondly $\mathcal{CP}$s sum up their own share and the ones received from others

$$\begin{aligned}
g(x) = \sum_{i=1}^{n} g_i(x) &= \sum_{i=1}^{n} 0 + \sum_{i=1}^{n} c_{i_1}x + \sum_{i=1}^{n} c_{i_2}x^2 + \cdots + \sum_{i=1}^{n} c_{i_{k-1}}x^{k-1} \\
&= 0 + c'_1x + c'_2x^2 + \cdots + c'_{k-1}x^{k-1}.
\end{aligned}$$

In the end, we get a secret-shared zero $[\![zero]\!]$. The coefficients are unknown to everyone as each $\mathcal{CP}_t$ only knows its own addends (c_{t_i}, $i \in \{1, k-1\}$). Hence, the shares are only known to their holders. Notice that this part of the protocol does not depend on the value we want to reshare and so it can be precomputed during the offline phase. It is also independent of the data type to use this for, as zero is represented the same way for booleans, unsigned and signed integers. Complexity-wise, secret-sharing a zero takes one round and $n(n-1)b$ bits of communication between $\mathcal{CP}$s.

Adding zero. Once we have a share for $[\![zero]\!]$, we can simply locally add it to our share of the secret value $[\![s]\!]$. In the following g represents the polynomial for secret-shared zero, f for $[\![s]\!]$ and h for $[\![s' = s + 0]\!]$.

$$\begin{aligned}
g(x) &= 0 + c'_1x + c'_2x^2 + \cdots + c'_{k-1}x^{k-1} \\
f(x) &= s + c_{f_1}x + c_{f_2}x^2 + \cdots + c_{f_{k-1}}x^{k-1} \\
h(x) &= g(x) + f(x) \\
&= 0 + s + c'_1x + c_{f_1}x + c_{f_2}x^2 + c'_2x^2 + \cdots + c'_{k-1}x^{k-1} + c_{f_{k-1}}x^{k-1} \\
&= s + c''_1x + c''_2x^2 + \cdots + c''_{k-1}x^{k-1}.
\end{aligned}$$

The result represents the same secret s with different coefficients. These coefficients are unknown to all $\mathcal{CP}$s as the coefficients of a secret-shared zero are unknown. This part of the protocol also clearly works on all data types, but $[\![zero]\!]$ must be secret-shared over the same field as is $[\![s]\!]$, otherwise we might end up with a different secret value. Say we have $k = n = 2, s = 10$ and we share zero as an 8-bit unsigned integer, then the following polynomials could be created with calculations for shares

$$\begin{array}{lll}
g(x) = 0 + 200x & f(x) = 10 + 3x & \\
g(1) = 200 & f(1) = 13 & h(1) = 213 \\
g(2) = 400 \bmod 251 = 149 & f(2) = 16 & h(2) = 165.
\end{array}$$

Now, depending on the type of s and s', we get

for 8-bit values	**for 16-bit values**
$213 = s' + c \pmod{251}$	$213 = s' + c \pmod{65521}$
$165 = s' + 2c \pmod{251}$	$165 = s' + 2c \pmod{65521}$
$48 = -c \pmod{251}$	$48 = -c \pmod{65521}$
$c = 203$	$c = 65473$
$s' = 10$	$s' = 261 \neq 10.$

Algorithm 5 summarises the protocol for resharing a secret-shared value. As the second part – adding zero – does not require any communication, the total complexity is one round and $n(n-1)b$ bits of data is transferred between computing parties. There are multiple reasons, why we might need to reshare our values, for instance before declassifying a value, as otherwise reusing that share somewhere else leaks information. For more information on the importance of resharing, see [Bog13, BLLP14]

Algorithm 5: Protocol for resharing a classified value `Reshare`($[\![s]\!]$)

```
Input: $[\![s]\!]$
Result: $[\![s']\!]$ with different shares, where $s = s'$
1 foreach computing party $\mathcal{CP}_t$ do
2     Classify(0)                                  // keeps $g_t(t)$ to oneself
3     Receive shares $g_i(t)$ from other $\mathcal{CP}$s
4     $[\![s']\!]_t = [\![s]\!]_t + \sum_{i=1}^{n} g_i(t)$
```

4.3 Declassification

After calculating a function using SMC on our secret-shared data, we may want to declassify the result. This result is also secret-shared with Shamir's secret sharing scheme. To reconstruct the secret, we first reshare the value and then everyone reveals their share. Published shares can be combined together using polynomial interpolation, see Section 2.1. In the formulae (2.1) and (2.2), t_i refers to the participating $\mathcal{CP}$s node numbers. Algorithm 6 describes the protocol for reconstructing a secret value.

This protocol has two rounds, however as mentioned before reshare can be precomputed and so we have only one online round. Communication costs between $\mathcal{CP}$s and $\mathcal{RP}$ are kb bits. Resharing required $n(n-1)b$ bits, however we only need the reshared shares for our k participating $\mathcal{CP}$s so we get the cost $k(k-1)b$ instead for the offline phase.

Algorithm 6: Protocol for reconstructing the secret `Declassify`($[\![s]\!]$)

```
Input: $[\![s]\!]$, threshold $k$, participating $\mathcal{CP}$s node numbers $t_i \in \{1, \dots, n\}$
Result: Secret $s$
1 foreach computing party $\mathcal{CP}_{t_i}$ do
2     $[\![s']\!] = \texttt{Reshare}([\![s]\!])$
3     $\mathcal{CP}_{t_i}$ sends $[\![s']\!]_{t_i}$ to $\mathcal{RP}$
4 $\mathcal{RP}$ calculates $s = \sum_{t_i} [\![s']\!]_{t_i} b_{t_i}$
```

If we want to use declassification inside other protocols so that all $\mathcal{CP}$s know the value, then the $\mathcal{RP}$ (one of the $\mathcal{CP}$s can act as the $\mathcal{RP}$) sends the values back, i.e. we would have two rounds and $kb + nb$. If we prioritise minimising the round count, then k $\mathcal{CP}$s can send their values to everyone and then all $\mathcal{CP}$s reconstruct themselves. This would lead to one round, but $(n-1)kb$ communication cost. It would be possible to do load balancing and have $k-1$ previous $\mathcal{CP}$s send their shares, however in that case we need to reshare all the shares and offline communication cost would be $n(n-1)b$.

Booleans as 8-bit unsigned integers run the same algorithm. Signed integers, on the other hand, require some post-processing. More precisely, we revert, what we did in Algorithm 4. After declassification, if $s > \frac{p}{2}$, i.e. it is a negative value represented as a large positive one, we subtract p, see Algorithm 7.

Algorithm 7: Protocol for declassifying a signed integer

```
Input: Prime $p$, $[\![s]\!]$
Result: Signed secret value $s$
1 $\mathcal{RP}$ runs $u = \texttt{Declassify}([\![s]\!])$
2 $\mathcal{RP}$ calculates:
3 if $u < \frac{p}{2}$ then
4     $s = u$
5 else
6     $s = u - p$
```

Chapter 5 Arithmetic protocols

In this section, we shall give protocols for addition, subtraction and multiplication. They will be described for unsigned integers, however they can be used to implement boolean arithmetic, as booleans were internally 8-bit unsigned integers. Signed integers also work without any extra effort due to their underlying representation.

5.1 Addition and subtraction with a public value

We can just add the public value to, or subtract from, each share. This protocol, see Algorithm 8, is done locally. We can see the additive inverse as the secret-shared value subtracted from zero, i.e. $-[\![s]\!] = 0 - [\![s]\!]$

Algorithm 8: Addition of a public value $[\![s]\!] + v$ (subtraction $[\![s]\!] - v$)

```
Input: $[\![s]\!]$, public value $v$
Result: $[\![r]\!]$, where $r = s + v$ (subtraction $r = s - v$)
1 foreach computing party $\mathcal{CP}_t$ do
2     compute $[\![r]\!]_t = [\![s]\!]_t + v$        // subtraction $[\![r]\!]_t = [\![s]\!]_t - v$
```

Theorem 1. The addition and subtraction of a public value protocols in Algorithm 8 are correct.

Proof. For correctness, we need to show that $r = s + v$ and $r = s - v$, correspondingly. Let $f(x)$ denote the polynomial for shares of s.

$$g(x) = f(x) + v = s + v + c_1x + c_2x^2 + \cdots + c_{k-1}x^{k-1}$$
$$g(x) = f(x) - v = s - v + c_1x + c_2x^2 + \cdots + c_{k-1}x^{k-1}$$

The polynomial $g(x)$ clearly represents shares of r and even the coefficients have not changed. □

Theorem 2. The addition and subtraction of a public value protocols in Algorithm 8 are perfectly simulatable against a passive adversary.

Proof. As there is no communication, the protocol run is perfectly simulatable. But the output shares depend on the input shares, hence it is not universally composable. □

5.2 Multiplication with a public value

We can just locally multiply each share with the public value, see Algorithm 9.

Algorithm 9: Multiplication with a public value $v[\![s]\!]$

```
Input: $[\![s]\!]$, public value $v$
Result: $[\![r]\!]$, where $r = vs$
1 foreach computing party $\mathcal{CP}_t$ do
2     compute $[\![r]\!]_t = [\![s]\!]_t v$
```

Theorem 3. The multiplication with a public value protocol in Algorithm 9 is correct.

Proof. For correctness, we need to show that $r = vs$. Let $f(x)$ denote the polynomial for shares of s.

$$\begin{aligned} g(x) = f(x) \cdot v &= sv + vc_1x + vc_2x^2 + \cdots + vc_{k-1}x^{k-1} \\ &= sv + c'_1x + c'_2x^2 + \cdots + c'_{k-1}x^{k-1}, \end{aligned}$$

The coefficients are changed for all shares, but we can see that polynomial $g(x)$ represents the Shamir secret-shared sv. □

Theorem 4. The multiplication with a public value protocol in Algorithm 9 is perfectly simulatable against a passive adversary.

Proof. Similarly to Theorem 2, the protocol is perfectly simulatable, but not universally composable. □

5.3 Addition and subtraction for two shared values

This simple local protocol is given in Algorithm 10.

Theorem 5. The addition and subtraction protocols in Algorithm 10 are correct.

Algorithm 10: Addition of two secret values $[\![q]\!] + [\![r]\!]$ (subtraction $[\![q]\!] - [\![r]\!]$)

```
Input: $[\![q]\!]$, $[\![r]\!]$
Result: $[\![s]\!]$, where $s = q + r$ (subtraction $s = q - r$).
1 foreach computing party $\mathcal{CP}_t$ do
2     compute $[\![s]\!]_t = [\![q]\!]_t + [\![r]\!]_t$        // subtraction $[\![s]\!]_t = [\![q]\!]_t - [\![r]\!]_t$
```

Proof. For correctness, we need to show that $s = q + r$ and $s = q - r$ correspondingly. Let $f_q(x)$ denote the polynomial for shares of q and $f_r(x)$ for shares of r.

$$\begin{aligned} f_q(x) &= q + c_{q1}x + c_{q2}x^2 + \cdots + c_{q\,k-1}x^{k-1} \\ f_r(x) &= r + c_{r1}x + c_{r2}x^2 + \cdots + c_{r\,k-1}x^{k-1} \\ g(x) = f_q(x) + f_r(x) &= q + r + c_{q1}c_{r1}x + c_{q2}c_{r2}x^2 + \cdots + c_{q\,k-1}c_{r\,k-1}x^{k-1} \\ &= q + r + c'_1x + c'_2x^2 + \cdots + c'_{k-1}x^{k-1} \end{aligned}$$

Again the coefficients are different, but $g(x)$ represents the secret-shared $q + r$. Subtraction works similarly. □

Theorem 6. The addition and subtraction protocols in Algorithm 10 are perfectly simulatable against a passive adversary.

Proof. Similarly to Theorem 2, the protocol is perfectly simulatable, but not universally composable. □

5.4 Multiplication of two shared values

Let $f_q(x)$ denote the polynomial for shares of q and $f_r(x)$ for shares of r. If we simply multiply the shares locally, we end up with a secret-shared polynomial f_{qr}, where $f_{qr}(0) = qr$, but the degree is $2(k-1)$:

$$\begin{aligned} f_{qr}(x) &= f_q(x)f_r(x) \\ &= (q + c_{q1}x + c_{q2}x^2 + \cdots + c_{q\,k-1}x^{k-1})(r + c_{r1}x + c_{r2}x^2 + \cdots + c_{r\,k-1}x^{k-1}) \\ &= qr + (qc_{r1} + rc_{q1})x + (qc_{r2} + rc_{q2})x^2 + \cdots + (qc_{r\,k-1} + rc_{q\,k-1})x^{k-1} \\ &\quad + (c_{q1}x + c_{q2}x^2 + \cdots + c_{q\,k-1}x^{k-1})(c_{r1}x + c_{r2}x^2 + \cdots + c_{r\,k-1}x^{k-1}) \\ &= qr + c'_1x + c'_2x^2 + ... + c'_{2(k-1)}x^{2(k-1)}. \end{aligned}$$

We need to somehow reduce the result to a polynomial with degree $k-1$, as otherwise the secrets are no longer using k-out-of-n, but $2k$-out-of-n threshold scheme. This is the reason why multiplication of two shares cannot be done locally. Note also that the polynomial we got is not a random one, hence we also need to perform randomization. Gennaro, Rabin and Rabin [GRR98] showed how we can achieve both in a single step. If we had a trusted entity, we could use the Lagrange's formula to declassify the multiplication result,

$$qr = f_{qr}(0) = \sum_{i=1}^{n} f_{qr}(t_i)b_i,$$

and then secret-share it to get shares for a polynomial with degree $k-1$,

$$[\![q]\!] \cdot [\![r]\!] = \texttt{Classify}(f_{qr}(0)).$$

Notice that we need to have at least $2k-1$ participants, i.e. $2k-1 \leq n$, to get something useful from f_{qr}. We can change the order of operations to eliminate the need for a trusted entity. Each computing party t_i first creates a random polynomial

$$f_{qr_i}(x) = f_{qr}(t_i)b_i + c_{1i}x + c_{2i}x^2 + ... + c_{k-1i}x^{k-1},$$

then shares it to other $\mathcal{CP}$s, i.e. sends $f_{qr_i}(x)$ to $\mathcal{CP}_x$. Finally, the $\mathcal{CP}$s add all their shares together. The full protocol is described in Algorithm 11. Multiplication consists of one round and the only network usage comes from every $\mathcal{CP}$ running the classification protocol. In total, the communication cost is $n(n-1)b$ bits.

Algorithm 11: Multiplication of two secret values $[\![q]\!] \cdot [\![r]\!]$

```
Input: $[\![q]\!]$, $[\![r]\!]$
Result: $[\![s]\!]$, where $s = qr$
1 foreach computing party $\mathcal{CP}_t$ do
2     $[\![z]\!]_t = [\![q]\!]_t [\![r]\!]_t b_i$
3     Classify($[\![z]\!]_t$)                    // keeps $f_{qr_t}(t)$ to oneself
4     Receive shares $f_{qr_i}(t)$ from other $\mathcal{CP}$s
5     $[\![s]\!]_t = \sum f_{qr_i}(t)$
```

Theorem 7. The multiplication protocol in Algorithm 11 is correct.

Proof. For correctness, we need to show that $s = qr$. The sum calculated as the result is

$$\begin{aligned}\sum_{i=1}^{n} f_{qr_i}(x) &= \sum_{i=1}^{n}(f_{qr}(t_i)b_i + c_{1i}x + c_{2i}x^2 + ... + c_{ni}x^{k-1}) \\ &= \sum_{i=1}^{n} f_{qr}(t_i)b_i + \sum_{i=1}^{n} c_{1i}x + \sum_{i=1}^{n} c_{2i}x^2 + ... + \sum_{i=1}^{n} c_{k-1i}x^{k-1} \\ &= qr + c'_1x + c'_2x^2 + ... + c'_{k-1}x^{k-1}.\end{aligned}$$

The polynomial $f_{qr}(x)$ clearly represents shares of qr. □

Theorem 8. The multiplication protocol in Algorithm 11 is secure against a passive adversary.

Proof sketch. Algorithm 11 is symmetric for all the parties, hence we only need to view one set of corrupt computing parties $\mathcal{CP}_{c_1}, \mathcal{CP}_{c_2}, ..., \mathcal{CP}_{c_{k-1}}$ (there were at most $k-1$ corrupt $\mathcal{CP}$s). In total, the adversary sees values $f_{qr_i}(t)$ for every $i \in \{1, ..., n\}$ and $t \in \{c_1, ..., c_{k-1}\}$, which are outputs from the classification algorithm and, hence, are uniformly distributed and independent of the private inputs (shares of q and r). We can build a perfect simulator by generating uniformly distributed values and using them as honest parties' outputs. The protocol is also universally composable as simulator does not rewind adversary and input and output shares are independent. Further details are out of the scope of this thesis.

$\square$

5.5 Boolean arithmetic

As booleans are represented as 8-bit unsigned integers, we can use the same algorithms for addition and multiplication.

Negation

$$\neg[\![b]\!] = 1 - [\![b]\!]$$

Using only subtraction makes negation a local operation.

Conjunction

$$[\![a]\!] \wedge [\![b]\!] = [\![a]\!] \cdot [\![b]\!]$$

Disjunction

$$[\![a]\!] \vee [\![b]\!] = [\![a]\!] + [\![b]\!] - [\![a]\!] \cdot [\![b]\!]$$

Exclusive disjunction

$$[\![a]\!] \oplus [\![b]\!] = [\![a]\!] + [\![b]\!] - 2([\![a]\!] \cdot [\![b]\!])$$

Conjunction, disjunction and exclusive disjunction use multiplication and some also local operations such as addition or multiplying with a public value. Using the multiplication protocol brings the complexity to one round and communication cost up to $n(n-1)b = 8n(n-1)$.

Theorem 9. The boolean arithmetic operations are correct.

Proof. Correctness follows directly from the operations definitions and correctness of addition, subtraction and multiplication shown before. □

Theorem 10. The boolean arithmetic operations are secure against a passive adversary.

Proof sketch. All the sub-protocols used are perfectly simulatable, hence the main protocols are also perfectly simulatable. Conjunction is just one multiplication, hence it is universally composable. Disjunction and exclusive disjunction protocols contain a universally composable multiplication in the output share calculation. The shares of $[\![c]\!] = [\![a]\!] \cdot [\![b]\!]$ are uniformly distributed values, that are independent from $[\![a]\!]_t$ and $[\![b]\!]_t$. Therefore, the disjunction result for $\mathcal{CP}_t$ becomes

$$[\![c]\!]_t = [\![a]\!]_t + [\![b]\!]_t - [\![r]\!]_t = [\![a]\!]_t + [\![r']\!]_t = [\![r'']\!]_t$$

where $[\![r']\!]_t, [\![r'']\!]_t$ are uniformly distributed and independent from $[\![a]\!]_t$ and $[\![b]\!]_t$. The actual proof is more detailed and involves explicit construction of the simulator. However, these technical details are out of the scope of this thesis.

Similarly, for exclusive disjunction, the resulting share for $\mathcal{CP}_t$ is

$$[\![c]\!]_t = [\![a]\!]_t + [\![b]\!]_t - 2[\![r]\!]_t = [\![a]\!]_t + [\![b]\!]_t - [\![r']\!]_t = [\![a]\!]_t + [\![r'']\!]_t = [\![r''']\!]_t$$

where $[\![r']\!]_t, [\![r'']\!]_t, [\![r''']\!]_t$ are uniformly distributed and independent from $[\![a]\!]_t$ and $[\![b]\!]_t$. Therefore, disjunction and exclusive disjunction are universally composable. □

Chapter 6

Comparison operations

The algorithms for equality and less-than use various sub-protocols, which may contain further sub-protocols. Figure 6.1 shows the dependencies between the building blocks for comparison operations.

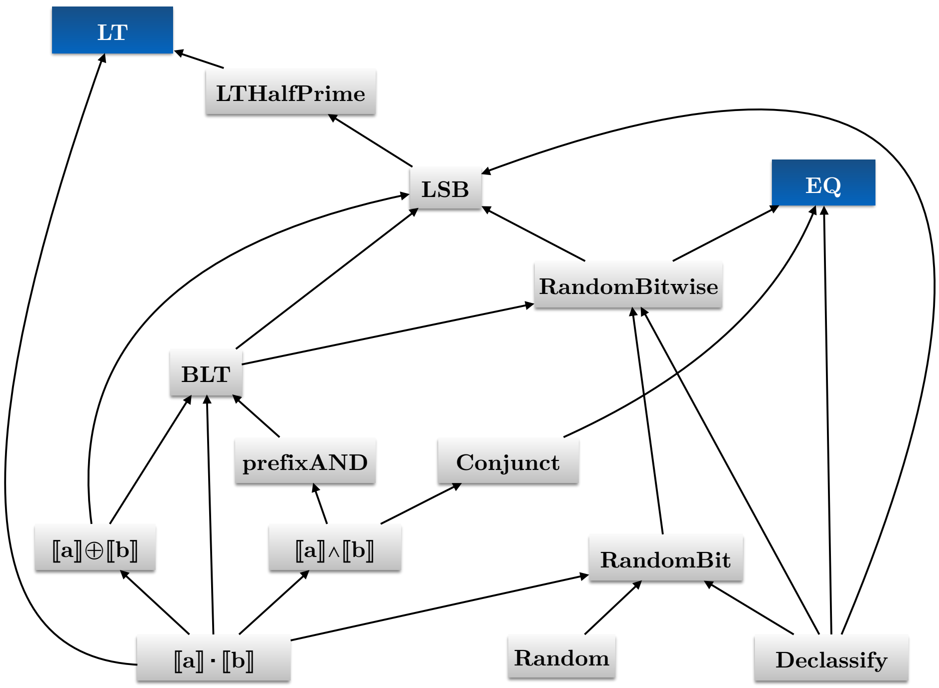

Figure 6.1: Protocol hierarchy

In the following protocols, let $\mathbf{a}_i$ represent the ith bit of a, i.e. $a = \sum_{i=0}^{\ell-1} 2^i \mathbf{a}_i$, where $\ell = b$ is the size of a in bits. Finding the ith bit for public values is trivial. Let the vector of a's bits $[\![\mathbf{a}]\!] = \{[\![\mathbf{a}_{\ell-1}]\!], ..., [\![\mathbf{a}_0]\!]\}$ represent a bitwise shared value. We can get bitwise secret-shared values by running the bit decomposition protocol (see Section 6.1.7) or bitwise sharing of a random number (see Section 6.1.8).

6.1 Sub-protocols

6.1.1 Secret-sharing a random value

Computing parties want to create a random value that no-one knows, but which is a Shamir secret shared value, i.e. k parties can reconstruct it. This can be done using Algorithm 12. First, every $\mathcal{CP}$ creates a random value $s_i \leftarrow F$ and classifies that, then all $\mathcal{CP}$s add their random shares together.

Algorithm 12: Secret-sharing a random value `Random()`

```
Input: Finite field $F$
Result: $[\![r]\!]$, where $r \in F$ is uniformly distributed
1 foreach computing party $\mathcal{CP}_t$ do
2     $s_t \overset{u}{\leftarrow} F$
3     Classify($s_t$)                          // keeps $f_t(t)$ to oneself
4     Receive shares $f_i(t)$ from other $\mathcal{CP}$s
5     $[\![r]\!]_t = \sum f_i(t)$
```

This protocol cannot be used for booleans as it does not guarantee the secret shared value to be in $\{0, 1\}$, see Section 6.1.2 for sharing a random bit. It does work for signed integers as all the functionality used (classification, addition) we have already described. Secret-sharing a random value requires one computation round using one classification per $\mathcal{CP}$. This results in total communication cost of $n(n-1)b$ bits. As there are no inputs, random number sharing can be done during the offline phase.

Theorem 11. The sharing a random value protocol in Algorithm 12 is correct.

Proof. To prove correctness, we need to show that r is a uniformly distributed secret-shared value. During the classification step, each $\mathcal{CP}_t$ creates a polynomial f_t,

$$\begin{aligned}
f_1(x) &= s_1 + c_{11}x + c_{12}x^2 + \cdots + c_{1k-1}x^{k-1} \\
f_2(x) &= s_2 + c_{21}x + c_{22}x^2 + \cdots + c_{2k-1}x^{k-1} \\
&\ldots \\
f_n(x) &= s_n + c_{n1}x + c_{n2}x^2 + \cdots + c_{nk-1}x^{k-1}.
\end{aligned}$$

Later, adding their own and all the random shares received together, we get

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} f_i(x) = \sum_{i=1}^{n} s_i + \sum_{i=1}^{n} c_{i1}x + \sum_{i=1}^{n} c_{i2}x^2 + \cdots + \sum_{i=1}^{n} c_{ik-1}x^{k-1} \\ &= \sum_{i=1}^{n} s_i + c'_1 x + c'_2 x^2 + \cdots + c'_{k-1}x^{k-1}. \end{aligned}$$

The resulting polynomial $f(x)$ clearly represents shares for a secret-shared value $[\![\sum_{i=1}^{n} s_i]\!]$. Adding together uniformly distributed values gives us a uniformly distributed value. □

Theorem 12. The sharing a random value protocol in Algorithm 12 is secure against a passive adversary.

Proof sketch. This sum $[\![\sum_{i=1}^{n} s_i]\!]$ is unknown to everyone, as each computing party only knows its own addend s_i. Hence, no-one knows the value of r. No matter who the corrupted parties are, the adversary only sees outputs from the classification algorithm and hence uniformly distributed values. We can build a perfect simulator by generating uniformly distributed values. As there are no input shares to depend on and each $\mathcal{CP}_t$ has their own uniformly distributed addend $f_t(t)$ that no-one else knows in their resulting share, the protocol is universally composable. □

6.1.2 Sharing a random bit

A straightforward way for getting a sharing of a random bit would be for all $\mathcal{CP}$s to share a random bit and then perform exclusive disjunction over those bits. There is, however a more efficient way described by Damgård et al. [DFK+06]. We start by randomly sharing a value $[\![r]\!]$. Next, we compute $[\![r]\!]^2$ using the multiplication protocol and declassify it. If $r^2 = 0$ we are unlucky and have to start over, otherwise we find its square root $r' = \sqrt{r^2}$, where $0 < r' < \frac{p}{2}$. Finally, we find the random bit by calculating $2^{-1}(r'^{-1}r + 1)$. The protocol is given in Algorithm 13.

In the complexity analysis, we see secret-sharing of a random value, multiplication and declassification. Note, that we do not need to reshare in the `Declassify` algorithm, as multiplication is universally composable. Excluding the small probability of $\frac{1}{|F|}$ having to start over, we get two rounds and communication cost of

$$n(n-1)b + n(n-1)b + k(n-1)b = (2n+k)(n-1)b.$$

As there are no inputs, these random bits can be precomputed.

Finding the square root $x = \sqrt{x^2} \bmod p$. For simplicity ,we look at only the case where $p \bmod 4 = 3$. This works for `uint8` and if we need to create bits for bigger types, then we can just convert up. Note, that, for complexity, we ignore this and assume we have an algorithm for finding square roots for any data type. For the restricted case, Cohen [Coh93] provides a simple formula

$$x = a^{(p+1)/4}.$$

Since we need $x < \frac{p}{2}$, then if we get $\frac{p}{2} < x < p$ we can use $-x \bmod p$ to get the x in the required range, i.e. we use $p - x$ to get the right value.

Algorithm 13: Sharing a random bit `RandomBit()`

```
Input: Finite field $F$
Result: $[\![b]\!]$, where $b \in \{0,1\}$ is a uniformly distributed bit
1 repeat
2     $[\![r]\!] = \texttt{Random}()$
3     $r2 = \texttt{Declassify}([\![r]\!] \cdot [\![r]\!])$
4 until $r2 \neq 0$
5 $r' = \sqrt{r2}$                                  // where $0 < r' < \frac{p}{2}$
6 $[\![b]\!] = 2^{-1}(r'^{-1}[\![r]\!] + 1)$
```

Theorem 13. The sharing a random bit protocol in Algorithm 13 is correct.

Proof. To prove correctness, we need to show that r is a uniformly distributed secret-shared bit. We have two cases:

$$r' = \begin{cases} r & \text{if } r < \frac{p}{2} \\ -r & \text{if } r \geq \frac{p}{2}. \end{cases}$$

The result $b = 2^{-1}(r'^{-1}r + 1)$ is then

$$b = \begin{cases} 2^{-1}(r^{-1}r + 1) & = 2^{-1}(1+1) \ = 1 \ \textbf{if } r < \dfrac{p}{2} \\ 2^{-1}((-r)^{-1}r + 1) & = 2^{-1}(-1+1) = 0 \ \textbf{if } r \geq \dfrac{p}{2}, \end{cases}$$

hence b is in $\{0,1\}$. Applying addition and multiplication with a (non-zero) public value to a uniformly distributed value (r) results in a uniformly distributed value. Uniformity of r additionally ensures that both cases from above are equally likely. □

Theorem 14. The sharing a random bit protocol in Algorithm 13 is secure against a passive adversary.

Proof sketch. First, we need to show that the published values do not leak any information. We have two cases. Either the declassification result $r2 = 0$, in which case we forget r and go to the beginning of the protocol, therefore there is no information to be leaked. In the other case $r2 \neq 0$, we learn that $r = \pm\sqrt{r2}$ is either r' or $-r'$, which is equally likely. We later get the final output bit based on which one of those it happened to be, as we only need this one bit of unknown and multiplication is universally composable, the declassification does not leak anything.

Secondly, no-one knows the value of b, because no-one knows r (Theorem 12). Thirdly, after the loop there are only local operations and, in repeat, all the sub-protocols used are perfectly simulatable. Hence, the main protocol is perfectly simulatable. As there are no input shares, random is universally composable and declassification does not leak anything, the protocol is universally composable. □

6.1.3 Conjunction of bits

The simplest, but not very efficient way to find the conjunction is to just conjunct bits one by one, see Algorithm 14. Disjunction or exclusive disjunction over all bits can be performed similarly.

Algorithm 14: Conjunction of bits `Conjunct`($[\![\mathbf{a}]\!]$)

```
**Input**: Bitwise shared value $[\![\mathbf{a}]\!]$
**Result**: $[\![b]\!]$, where $b \in \{0,1\}$ is the conjunction of $a$'s bits, i.e. $b = \wedge_{i=0}^{\ell-1}\mathbf{a}_i$
1 $[\![b]\!] = [\![\mathbf{a}_0]\!]$
2 **for** $i = 0$ **to** $\ell - 1$ **do**
3     $[\![b]\!] = [\![b]\!] \wedge [\![\mathbf{a}_i]\!]$
```

It clearly works for any data type, however it would be unreasonable to run it on booleans, as all the bits except the least significant bit are zero. Conjunction is implemented via multiplication. This gives our naive solution the complexity of $\ell - 1 = b - 1$ rounds and $(b-1)n(n-1)b \approx n(n-1)b^2$ bits to transfer. It is trivial to make this protocol better by doing operations in parallel and thus having $\log_2 \ell$ rounds with the same communication cost.

Theorem 15. The conjunction of bits protocol in Algorithm 14 is correct.

Proof. The result of this algorithm gives us

$$\begin{aligned} b &= (((\mathbf{a}_0 \wedge \mathbf{a}_1) \wedge \mathbf{a}_2) \wedge \cdots \wedge \mathbf{a}_{\ell-2}) \wedge \mathbf{a}_{\ell-1} \\ &= \mathbf{a}_0 \wedge \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \cdots \wedge \mathbf{a}_{\ell-2} \wedge \mathbf{a}_{\ell-1}. \end{aligned}$$

which is clearly conjunction of all bits. □

Theorem 16. The conjunction of bits protocol in Algorithm 14 is secure against a passive adversary.

Proof. Conjunction is universally composable, hence the conjunction of bits is also universally composable. □

6.1.4 Prefix-AND

The simplest, but not very efficient way to find prefix-AND is to just conjunct bits one by one, keeping the intermediate values, see Algorithm 15. Prefix-OR and prefix-XOR work analogically. Complexity is similar to conjunction of bits, $b-1$ rounds and $n(n-1)b(b-1) \approx n(n-1)b^2$ bits for communication.

Algorithm 15: Prefix-AND of bits `PrefixAND`($[\![\mathbf{a}]\!]$)

```
Input: Bitwise shared value $[\![\mathbf{a}]\!]$
Result: $[\![\mathbf{b}]\!]$, where $\mathbf{b}_i \in \{0,1\}$ and $\mathbf{b}_i = \wedge_{j=i}^{\ell-1}\mathbf{a}_j$ for all $i < \ell$
1 $[\![\mathbf{b}_{\ell-1}]\!] = [\![\mathbf{a}_{\ell-1}]\!]$
2 for $i = \ell - 2$ to $0$ do
3     $[\![\mathbf{b}_i]\!] = [\![\mathbf{b}_{i+1}]\!] \wedge [\![\mathbf{a}_i]\!]$
```

Theorem 17. The prefix-AND protocol in Algorithm 15 is correct.

Proof. The algorithm gives us

$$\begin{aligned}
\mathbf{b}_{\ell-1} &= \mathbf{a}_{\ell-1} \\
\mathbf{b}_{\ell-2} &= \mathbf{b}_{\ell-1} \wedge \mathbf{a}_{\ell-2} \\
&= \mathbf{a}_{\ell-2} \wedge \mathbf{a}_{\ell-1} \\
\mathbf{b}_{\ell-3} &= \mathbf{a}_{\ell-3} \wedge \mathbf{a}_{\ell-2} \wedge \mathbf{a}_{\ell-1} \\
&\dots \\
\mathbf{b}_2 &= \mathbf{a}_2 \wedge \dots \wedge \mathbf{a}_{\ell-2} \wedge \mathbf{a}_{\ell-1} \\
\mathbf{b}_1 &= \mathbf{b}_2 \wedge \mathbf{a}_1 \\
&= \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \dots \wedge \mathbf{a}_{\ell-2} \wedge \mathbf{a}_{\ell-1} \\
\mathbf{b}_0 &= \mathbf{b}_1 \wedge \mathbf{a}_0 \\
&= \mathbf{a}_0 \wedge \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \dots \wedge \mathbf{a}_{\ell-2} \wedge \mathbf{a}_{\ell-1},
\end{aligned}$$

which is clearly exactly what we want. □

Theorem 18. The prefix-AND protocol in Algorithm 15 is secure against a passive adversary.

Proof. Similarly to Theorem 16, the protocol is universally composable thanks to the composition of sub-protocols. □

6.1.5 Less-than for bitwise secret-shared values

We have two bitwise secret-shared values $[\![\mathbf{a}]\!]$ and $[\![\mathbf{b}]\!]$ and we want to compute $a \overset{?}{<} b$. Idea of this protocol is to find the most significant different bit. This can be done by, first, finding the exclusive disjunction. Then prefix-OR on the result and then subtracting each bit from the previous one. For example, e only has set the most significant bit that a and b differ by. Knowing that bit, we can multiply it with one of the values and determine whether that bit is set.

equation	bits
a	00001011
b	00001101
$c = a \oplus b$	00000110
$\mathbf{d}_i = \vee_{j=i}^{\ell-1} \mathbf{c}_j$	00000111
$\mathbf{e}_i = \mathbf{d}_i - \mathbf{d}_{i+1}$	00000100

equation	bits
b	00001101
e	00000100
$\mathbf{y}_i = \mathbf{a}_i \cdot \mathbf{b}_i$	00000100
$\sum \mathbf{y}_i$	1

The Algorithm 16 is only designed for unsigned integers. With booleans, it simply does not make sense and it would be enough to compare only the least significant bit. For signed integers, it would be easier with the modified two's complement version, but it is possible to make it work for centering around zero too. However, as we did not need bitwise less-than on signed integers in practice, we omit the details.

Algorithm 16: Less-than for bitwise secret-shared values BLT($[\![\mathbf{a}]\!], [\![\mathbf{b}]\!]$)

```
Input: Bitwise shared values $[\![\mathbf{a}]\!], [\![\mathbf{b}]\!]$
Result: $[\![x]\!]$, where $x \in \{0,1\}$ and $x = (a \overset{?}{<} b)$
1 $[\![\mathbf{c}]\!] = [\![\mathbf{a}]\!] \oplus [\![\mathbf{b}]\!]$
2 $[\![\mathbf{d}]\!] = \texttt{PrefixOR}([\![\mathbf{c}]\!])$
3 $[\![\mathbf{e}_{\ell-1}]\!] = [\![\mathbf{d}_{\ell-1}]\!]$
4 $[\![\mathbf{e}_i]\!] = [\![\mathbf{d}_i]\!] - [\![\mathbf{d}_{i+1}]\!]$ for all $i < \ell - 1$
5 $[\![x]\!] = \sum_{i=0}^{\ell-1}([\![\mathbf{e}_i]\!] \cdot [\![\mathbf{b}_i]\!])$
```

Analysing complexity, we can go line by line. First, the calculation of c requires $\ell = b$ parallel executions of the exclusive disjunction protocol, which gives us one round and $n(n-1)b^2$ bits to transfer. Second, we use prefix-OR for d, gaining $b-1$ rounds and $n(n-1)b(b-1)$ communication cost. After some local operations, we perform multiplication in parallel over all bits and get another round and $n(n-1)b^2$.

In total, that makes $b+1$ rounds and

$$n(n-1)b^2 + n(n-1)b(b-1) + n(n-1)b^2 = n(n-1)b(3b-1) \approx 3n(n-1)b^2$$

bits of communication.

Theorem 19. The bitwise less-than protocol in Algorithm 16 is correct.

Proof. We have three cases: $a < b$, $a > b$ or $a = b$, see corresponding Tables 6.1, 6.2 and 6.3. Assume, that x is the most significant different bit if the values differ. If the columns are merged, then there has to be the same value in that position in the variables. The symbol ? denotes either 1 or 0. Clearly, only in the case of $a < b$, we have a bit set in eb and hence $x = 1$.

	Bit in position				
Variable	$(\ell-1)$	$(\ell-2)$		1	0
a	?	?	...	?	?
b					
c	0	0		0	0
d	0	0		0	0
e	0	0		0	0
eb	0	0		0	0

Table 6.1: Bitwise less-than execution for $a = b$

	Bit in position								
Variable	$(\ell-1)$	$(\ell-2)$		$(x+1)$	x	$(x-1)$		1	0
a	?	?	...	?	0	?	...	?	?
b					1	?		?	?
c	0	0		0	1	?		?	?
d	0	0		0	1	1		1	1
e	0	0		0	1	0		0	0
eb	0	0		0	1	0		0	0

Table 6.2: Bitwise less-than execution for $a < b$

□

Theorem 20. The bitwise less-than protocol in Algorithm 16 is secure against a passive adversary.

	Bit in position								
Variable	$(\ell-1)$	$(\ell-2)$		$(x+1)$	x	$(x-1)$		1	0
a	?	?	...	?	1	?	...	?	?
b					0	?		?	?
c	0	0		0	1	?		?	?
d	0	0		0	1	1		1	1
e	0	0		0	1	0		0	0
eb	0	0		0	0	0		0	0

Table 6.3: Bitwise less-than execution for $a > b$

Proof sketch. All the sub-protocols are perfectly simulatable, hence this protocol is too. It is also universally composable, as the resulting shares come from the summation of universally composable multiplication protocol outputs. □

6.1.6 Bit composition

Sometimes, we have a bitwise secret-shared value and we want to get the secret-shared value itself. We can do that for unsigned integers by locally calculating

$$r = \sum_{i=0}^{\ell-1} 2^i \mathbf{r}_i,$$

see Algorithm 17. Boolean bit composition would mean $r = \mathbf{r}_0$. For signed integers it simply works, because internally we have them as unsigned integers. For example, for centering around zero representation, if the summation of bits would add up to more than $\lfloor \frac{p}{2} \rfloor$, then it is seen as a negative value and the declassification for signed integers (see Algorithm 7) would subtract p.

Algorithm 17: Bit composition `BitComposition`($[\![\mathbf{a}]\!]$)

```
  Input: Bitwise shared value $[\![\mathbf{a}]\!]$
  Result: $[\![a]\!]$, where $\mathbf{a}_i \in \{0,1\}$ represent the bits of $a$
1 $[\![a]\!] = \sum_{i=0}^{\ell-1} 2^i [\![\mathbf{a}_i]\!]$
```

Theorem 21. The bit composition protocol in Algorithm 17 is correct.

Proof. Correctness follows directly from the correctness of multiplication with public value (Theorem 3) and addition (Theorem 5). □

Theorem 22. The bit composition in Algorithm 17 is perfectly simulatable against a passive adversary.

Proof. Similarly to Theorem 2, no communication but the outputs depend on the input shares, therefore the protocol is only perfectly simulatable. □

6.1.7 Bit decomposition

We have a secret-shared value $[\![a]\!]$ and we want to get its secret-shared bits $[\![\mathbf{a}]\!] = \{[\![\mathbf{a}_{\ell-1}]\!], ...[\![\mathbf{a}_0]\!]\}$, where $\mathbf{a}_i$ denotes the i-th bit of a, i.e. $a = \sum_0^{\ell-1} 2^i \mathbf{a}_i$. During this work we did not implement the bit decomposition protocol `BitDecomposition`($[\![a]\!]$), however we use it in some of the algorithms to show alternatives.

6.1.8 Bitwise sharing of a random number

We generate each bit separately in parallel and check that it is in bounds, i.e. compute bitwise less-than $r < p$, reveal it. If it is false, then retry. The protocol is given in Algorithm 18. It works for both signed and unsigned integers as both of them have the same number of bits and internal representation. To create random booleans we would simply use the `RandomBit` protocol. We can use bit composition from Section 6.1.6 to also find $[\![r]\!]$, which does not change the complexity.

There are no input shares, so we can generate bitwise random numbers during the offline phase. Complexity-wise, we ignore the $1 - \frac{|Z_p|}{|Z_{2^b}|}$ probability for reruns. We do not need to reshare in declassify as c comes from the universally composable bitwise less-than protocol. There are the first two rounds, where we create the random bits in parallel and $b + 1$ rounds for bitwise less-than. The declassification adds another round, which makes total $b + 4$. Data communication adds up to

$$\begin{aligned}
&(2n + k)(n - 1)b^2 + n(n - 1)b(3b - 1) + k(n - 1)b \\
&= (n - 1)b((2n + k)b + n(3b - 1) + k) \\
&= (n - 1)b(5nb + kb - n + k) \\
&\approx (n - 1)b(5nb + kb) \approx (n - 1)(5n + k)b^2.
\end{aligned}$$

Algorithm 18: Bitwise sharing of a random number `RandomBitwise()`

```
    Input: Prime $p$, bit-length $\ell$
    Result: $[\![\mathbf{r}]\!]$, where $r$ is uniformly distributed, $\mathbf{r}_i \in \{0,1\}$ and $r = \sum_{i=0}^{\ell-1} 2^i \mathbf{r}_i$
1   repeat
2       $[\![\mathbf{r}_i]\!]$ = RandomBit() for all $i < \ell$
3       $[\![c]\!]$ = BLT($[\![\mathbf{r}]\!], p$)
4       Declassify($[\![c]\!]$)
5   until $c = 1$
```

Theorem 23. The bitwise sharing of a random number protocol in Algorithm 18 is correct.

Proof. If we would just create random bits for each position, then we would clearly get a uniformly distributed number in Z_{2^ℓ}. On that uniformly distributed number, we use rejection sampling and hence get a uniformly distributed value in Z_p. □

Theorem 24. The bitwise sharing of a random number protocol in Algorithm 18 is secure against a passive adversary.

Proof sketch. Firstly, lets look at the declassification. As there are no input shares it could only leak information about the output shares. As a result from bitwise less-than protocol c is a boolean value. We can ignore the case, when it is false, as then we would throw away all the work we did and start from the beginning. If $c = 1$ we learned that $r < p$, which does not say anything about $r \in Z_p$ as the maximum value r can have is $p - 1$.

All the sub-protocols are perfectly simulatable, hence this protocol is too. The output $[\![\mathbf{r}]\!]$ comes from a universally composable protocol and is used only in a universally composable protocol, hence this protocol is universally composable. □

6.1.9 Least significant bit

The idea for least significant bit came from [NO07], see Algorithm 19, where

$$\mathbf{c}_0 \oplus [\![\mathbf{r}_0]\!] = \begin{cases} [\![\mathbf{r}_0]\!] & \text{if } \mathbf{c}_0 = 0 \\ 1 - [\![\mathbf{r}_0]\!] & \text{if } \mathbf{c}_0 = 1. \end{cases}$$

Algorithm 19: Least significant bit `LSB`($[\![a]\!]$)

```
Input: $[\![a]\!]$
Result: $[\![b]\!]$, where $b \in \{0,1\}$ and $b = \mathbf{a}_0$
1 $[\![\mathbf{r}]\!]$ = RandomBitwise()
2 $[\![r]\!]$ = BitComposition($[\![\mathbf{r}]\!]$)
3 $[\![c]\!] = [\![a]\!] + [\![r]\!]$
4 $c$ = Declassify($[\![c]\!]$)
5 $[\![x]\!] = \mathbf{c}_0 \oplus [\![\mathbf{r}_0]\!]$
6 $[\![y]\!]$ = BLT($c$, $[\![\mathbf{r}]\!]$)
7 $[\![b]\!] = [\![x]\!] \oplus [\![y]\!]$
```

It works on the internal representation and hence can be used for all data types. Communication-demanding sub-protocols are bitwise random number sharing, declassification, bitwise less-than and exclusive disjunction. Bitwise random number sharing and resharing from declassify can be done in parallel during precomputation.

This gives us a round count of $b + 4$ and data communication cost of

$$\begin{aligned}
&(n-1)b(5nb + kb - n + k) + k(k-1)b \\
&= (5n^2b + nkb - n^2 + nk - 5nb - kb + n - 2k + k^2)b \\
&= ((5n^2 + nk - 5n - k)b - n^2 + nk + k^2 + n - 2k)b \\
&= ((5n + k)(n-1)b - n^2 + nk + k^2 + n - 2k)b \\
&\approx (n-1)(5n + k)b^2.
\end{aligned}$$

In the online phase, we have $1 + (b + 1) + 1 = b + 3$ rounds and communication

$$\begin{aligned}
&k(n-1)b + n(n-1)b(3b-1) + n(n-1)b \\
&= (n-1)(k + 3nb)b \\
&\approx 3n(n-1)b^2.
\end{aligned}$$

Theorem 25. The least significant bit protocol in Algorithm 19 is correct.

Proof. For correctness, we need to show, that $b = \mathbf{a}_0$. The least significant bit of $c = a + r$ is

$$\mathbf{c}_0 = \begin{cases} \mathbf{a}_0 \oplus \mathbf{r}_0 & \text{if } c \geq r \\ \mathbf{a}_0 \oplus \mathbf{r}_0 \oplus 1 & \text{if } c < r, \end{cases}$$

as only an overflow would make $c < r$ true, in which case p is subtracted from c. As p is odd, it flips the least significant bit. Then we get

$$\begin{aligned}
b = x \oplus y &= \mathbf{c}_0 \oplus \mathbf{r}_0 \oplus (c \overset{?}{<} r) \\
&= \begin{cases} \mathbf{c}_0 \oplus \mathbf{r}_0 \oplus 0 & \text{if } c \geq r \\ \mathbf{c}_0 \oplus \mathbf{r}_0 \oplus 1 & \text{if } c < r \end{cases} \\
&= \begin{cases} \mathbf{a}_0 \oplus \mathbf{r}_0 \oplus \mathbf{r}_0 & \text{if } c \geq r \\ (\mathbf{a}_0 \oplus \mathbf{r}_0 \oplus 1) \oplus \mathbf{r}_0 \oplus 1 & \text{if } c < r \end{cases} \\
&= \mathbf{a}_0.
\end{aligned}$$

□

Theorem 26. The least significant bit protocol in Algorithm 19 is secure against a passive adversary.

Proof sketch. We need to show, that declassification does not leak any information. The value r comes from a universally composable `RandomBitwise`, but $[\![\mathbf{r}]\!]$ and $[\![r]\!]$ depend on each other. We declassify $a + r$, where r is a random uniformly distributed value, hence $a + r$ is a random uniformly distributed value. Unless we reveal r later, we are safe. The only place r is later used non-locally is in the bitwise less-than protocol, which is universally composable and therefore does not leak information about r. Further details are out of the scope of this thesis.

All the sub-protocols used are perfectly simulatable and the resulting shares come from universally composable exclusive disjunction, hence the protocol is universally composable. □

6.1.10 Comparison to half prime for unsigned integers

Something multiplied with two always has a least significant bit zero. However, if there was an overflow, then the least significant bit will be one as p was subtracted. Hence, we can find $(a < \frac{p}{2}) = \texttt{LSB}(2a \bmod p)$. We only need to find the least significant bit. Therefore, the round and communication complexities are equal. Boolean values are clearly always less than half of the prime. For signed integers, the Algorithm 20 finds the comparison to their internal representation. That is handy for the centering around zero representation, where it shows if the value is positive or negative.

```
Algorithm 20: Compare to half prime LTHalfPrime($[\![a]\!]$)
  Input: $[\![a]\!]$
  Result: $[\![b]\!]$, where $b \in \{0,1\}$ and $b = (a \overset{?}{<} \frac{p}{2})$
1 $[\![b]\!] = \texttt{LSB}(2[\![a]\!])$
```

Theorem 27. The comparison to half prime protocol in Algorithm 20 is correct.

Proof. To prove correctness, we need to show that b is zero if $a < \frac{p}{2}$ and one otherwise. We have two cases:

$$\begin{aligned} a \in \{0, ..., \frac{p-1}{2}\} &\iff \texttt{LSB}(2a \bmod p) = \texttt{LSB}(2a) = 0 \\ a \in \{\frac{p-1}{2} + 1, ..., p-1\} &\iff \texttt{LSB}(2a \bmod p) = \texttt{LSB}(2a - p) = 1. \end{aligned}$$

□

Theorem 28. The comparison to half prime protocol in Algorithm 20 is secure against a passive adversary.

Proof. The least significant bit protocol is universally composable, hence this protocol is universally composable too. □

6.2 Equality

The first thing to notice about equality is that the problems $[\![a]\!] \overset{?}{=} [\![b]\!]$ and $[\![a]\!]-[\![b]\!] \overset{?}{=} 0$ are equivalent. Hence, now we just need to check if the value is zero $[\![z]\!] \overset{?}{=} 0$. Also, we need to implement equality only on the underlying unsigned data type, as two booleans or signed integers are equal iff their underlying unsigned values are equal.

6.2.1 Equality with a public result

Finding out equality to zero is easier, when the result can be public. We can share a random value r using Algorithm 12 and multiply it with $[\![z]\!]$. Now we declassify $[\![r]\!] \cdot [\![z]\!]$. This does not leak any information as r is an unknown value. If $rz \neq 0$, then $z \neq 0$. Otherwise, either z and/or r was zero. We can now declassify r. As the multiplication result was zero, we do not leak any information about z other than $z \overset{?}{=} 0$. If we get $r = 0$, we just choose another random value and repeat the algorithm, as we were unlucky and learned nothing ($0z = 0$). The protocol is given in Algorithm 21 and its time complexity is probabilistic. We need to start over only if our secret-shared random value happened to be zero. The value is uniformly chosen from F, therefore the probability of going to the beginning of the loop is $\frac{1}{|F|}$. Note, that without revealing r, we could find equality with probability $\frac{|F|-1}{|F|}$ (which can be increased by performing it multiple times).

Complexity-wise, we again ignore the rerun probability. Therefore, we have a random number sharing, multiplication and two declassifications in the worst case. Note, that resharing is not needed in declassifications and so only random is generated in the offline phase. For the online phase, we get three rounds and

$$n(n-1)b + 2k(n-1)b = (n+2k)(n-1)b$$

bits communication cost.

Theorem 29. The equality with a public result protocol in Algorithm 21 is correct.

Proof. To prove correctness, we need to show that e is true if $a = b$ and false otherwise. In the first case, $z = 0$, next $rz = 0$. Finally, when $r \neq 0$, which with probably $\frac{|F|-1}{|F|}$ happens on the first run (if not, then some later loop execution), then true is returned. When $a \neq b$, we get $z \neq 0$ and $rz \neq 0$, which return false. Multiplication can only result in zero, if either or the values are zero, because we do not have zero divisors in our prime groups, e.g. for `uin8` we do not have any values that $zr = 251$. □

Theorem 30. The equality with a public result protocol in Algorithm 21 is secure against a passive adversary.

Algorithm 21: Equality with a public result `EQPublic`$(\llbracket a \rrbracket, \llbracket b \rrbracket)$

```
Input: $\llbracket a \rrbracket, \llbracket b \rrbracket$
Result: $e = (a \overset{?}{=} b)$
$\llbracket z \rrbracket = \llbracket a \rrbracket - \llbracket b \rrbracket$
while true do
    $\llbracket r \rrbracket$ = Random()
    if Declassify($\llbracket r \rrbracket \cdot \llbracket z \rrbracket$) $\neq 0$ then
        return false
    else if Declassify($\llbracket r \rrbracket$) $\neq 0$ then
        return true
    // Else we didn't learn anything as $0z = 0$
```

Proof sketch. Firstly, we need to show, that published values do not leak any information except the result of equality testing. Since r is a uniformly distributed random value and all operations are in a finite field, declassification of rz does not reveal anything about z. Lets examine the case, when we reach the second declassification. If $a = b$ then $rz = 0$ otherwise if $a \neq b$ it is a random field element. If $r = 0$ then we learn nothing. If $r \neq 0$ then $z = 0$, which can hold only if $a = b$. Consequently the distribution of published results can be efficiently simulated knowing only the output $a = b$.

Subtraction is perfectly simulatable and the other sub-protocols used are universally composable, therefore this protocol leaks nothing beyond public outputs. Since public outputs can be simulated by knowing the output, this protocol is universally composable. □

6.2.2 Equality with bit decomposition

Damgård et al. [DFK+06] describe one way to check if a value is zero by performing a bit decomposition and then finding the conjunction of negations of all those bits (or negation of bits disjunction), see Algorithm 22. The total complexity becomes the added complexity of bit decomposition and disjunction of bits. As we did not implement bit decomposition, this protocol is also not implemented in this work.

Theorem 31. The equality with bit decomposition protocol in Algorithm 22 is correct.

Proof. To prove correctness, we need to show that e is one, if $a = b$ and zero,

Algorithm 22: Equality with bit decomposition `EQbd`($[\![a]\!], [\![b]\!]$)

```
Input: $[\![a]\!], [\![b]\!]$
Result: $[\![e]\!]$, where $e = (a \overset{?}{=} b)$
1 $[\![z]\!] = [\![a]\!] - [\![b]\!]$
2 $[\![\mathbf{z}]\!] = \texttt{BitDecomposition}([\![z]\!])$
3 $[\![e]\!] = \neg\texttt{Disjunct}([\![\mathbf{z}]\!])$
```

otherwise. In the first case,

$$\begin{aligned} a &= b \\ z &= 0 \\ \mathbf{z}_i &= 0 \qquad \text{for all } i < l \\ e &= \neg \textstyle\bigvee_{i=0}^{\ell-1} (\mathbf{z}_i) = \neg \bigvee_{i=0}^{\ell-1} (0) = \neg 0 = 1. \end{aligned}$$

When the values are not equal,

$$\begin{aligned} a &\neq b \\ z &\neq 0 \\ \exists \mathbf{z}_i &\neq 0 \\ e &= \neg \textstyle\bigvee_{i=0}^{\ell-1} (\mathbf{z}_i) = \neg(... \vee 1 \vee ...) = \neg 1 = 0. \end{aligned}$$

□

6.2.3 Equality without bit decomposition

Performing bit decomposition is not very efficient for Shamir's secret sharing. Therefore, Nishide and Ohta [NO07] created a simpler algorithm by randomizing z and checking if it is equal to the random value used. We start by sharing bits for a random value $[\![\mathbf{r}]\!]$, then add it to $[\![z]\!]$ and declassify the result $[\![c]\!]$. Next, we check the equality of each bit for c and r. Finally, we perform conjunction over all those bits. The protocol is given in Algorithm 23, where

$$(\mathbf{c}_i \overset{?}{=} [\![\mathbf{r}_i]\!]) = \begin{cases} [\![\mathbf{r}_i]\!] & \text{if } \mathbf{c}_i = 1 \\ 1 - [\![\mathbf{r}_i]\!] & \text{if } \mathbf{c}_i = 0. \end{cases}$$

This protocol allows some work to be done offline, namely bitwise sharing of a random number and resharing from declassify. As it is done offline, it is clearly independent and can be done in parallel, making the round count $\texttt{max}(b+4, 1) = b+4$ and the communication cost (same as `LSB`, see Section 6.1.9)

$$\begin{aligned} &(n-1)b((5n+k)b - n + k) + k(k-1)b \\ &\approx (n-1)(5n+k)b^2. \end{aligned}$$

The online phase requires $b - 1 + 1 = b$ rounds and the communication cost is

$$k(n-1)b + n(n-1)(b-1)b = (n-1)(k + nb - n)b \approx (n-1)nb^2.$$

Algorithm 23: Equality without bit decomposition $\texttt{EQ}([\![a]\!], [\![b]\!])$

```
Input: $[\![a]\!]$, $[\![b]\!]$, bit-length $\ell$
Result: $[\![e]\!]$, where $e = (a \overset{?}{=} b)$
1 $[\![z]\!] = [\![a]\!] - [\![b]\!]$
2 $[\![\mathbf{r}]\!] = \texttt{RandomBitwise}()$
3 $[\![r]\!] = \texttt{BitComposition}([\![\mathbf{r}]\!])$
4 $[\![c]\!] = [\![z]\!] + [\![r]\!]$
5 $c = \texttt{Declassify}([\![c]\!])$
6 $[\![\mathbf{x}_i]\!] = (\mathbf{c}_i \overset{?}{=} [\![\mathbf{r}_i]\!])$ for all $i < \ell$
7 $[\![e]\!] = \texttt{Conjunct}([\![\mathbf{x}]\!])$
```

Theorem 32. The equality without bit decomposition protocol in Algorithm 23 is correct.

Proof. To prove correctness, we need to show that e is one if $a = b$ and zero otherwise. In the first case

$$\begin{aligned} a &= b \\ z &= 0 \\ c &= z + r = r \\ \mathbf{x}_i &= (\mathbf{c}_i \overset{?}{=} \mathbf{r}_i) = 1 \qquad \text{for all } i < l \\ e &= \wedge_{i=0}^{\ell-1}(\mathbf{x}_i) = \wedge_{i=0}^{\ell-1}(1) = 1. \end{aligned}$$

When the values are not equal

$$\begin{aligned} a &\neq b \\ z &\neq 0 \\ c &= z + r \neq r \\ \mathbf{x}_i &= (\mathbf{c}_i \overset{?}{=} \mathbf{r}_i) \qquad \text{for all } i < l \\ &\exists(\mathbf{c}_i \neq \mathbf{r}_i) \iff \exists \mathbf{x}_i \neq 0 \\ e &= \wedge_{i=0}^{\ell-1}(\mathbf{x}_i) = \ldots \wedge 0 \wedge \ldots = 0. \end{aligned}$$

□

Theorem 33. The equality without bit decomposition protocol in Algorithm 23 is secure against a passive adversary.

Proof. All the sub-protocol are perfectly simulatable and the resulting shares come from conjunction, which is universally composable, hence this protocol is universally composable too. □

6.3 Less-than

For less-than comparison, we can again use bit decomposition [DFK+06], but a better option was discovered by Nishide and Ohta [NO07], who used comparison to $\frac{p}{2}$. For the boolean datatype, less-than comparison does not make sense, so we do not consider it at all.

6.3.1 Less-than with bit decomposition

Once we have the bitwise sharing of values, we can simply use bitwise less than comparison, see Algorithm 24. Total complexity becomes the added complexity of two parallel bit decompositions and a bitwise less-than protocol. As we did not implement bit decomposition, this protocol is also not implemented during this work.

Algorithm 24: Less-than with bit decomposition `LTbd`($[\![a]\!], [\![b]\!]$)

```
Input: $[\![a]\!]$, $[\![b]\!]$
Result: $[\![e]\!]$, where $e = (a \overset{?}{<} b)$
1 $[\![\mathbf{a}]\!] = \texttt{BitDecomposition}([\![a]\!])$
2 $[\![\mathbf{b}]\!] = \texttt{BitDecomposition}([\![b]\!])$
3 $[\![e]\!] = \texttt{BLT}([\![\mathbf{a}]\!], [\![\mathbf{b}]\!])$
```

Theorem 34. The less-than with bit decomposition protocol in Algorithm 24 is correct for unsigned integers.

Proof. Correctness follows directly from correctness of the bitwise less-than protocol, which gives us one iff $a < b$. □

6.3.2 Less than without bit decomposition

We first find $w = a < \frac{p}{2}$, $x = b < \frac{p}{2}$ and $y = a - b < \frac{p}{2}$. Then it is possible to calculate less-than, see Tables 6.4 and 6.5 for unsigned and signed values respectively. Merged cells indicate equal value and question mark in the tables represents either zero or one.

$a < \frac{p}{2}$	$b < \frac{p}{2}$	$a - b < \frac{p}{2}$	$a < b$
1	0	?	1
0	1	?	0
?		0	1
?		1	0

Table 6.4: Less-than for unsigned integers

$a < \frac{p}{2}$	$b < \frac{p}{2}$	$a - b < \frac{p}{2}$	$a < b$
1	0	?	**0**
0	1	?	**1**
?		0	1
?		1	0

Table 6.5: Less-than for signed integers in centered around zero representation

For unsigned integers, our formula becomes

$$\begin{aligned} e &= w(1-x) + wx(1-y) + (1-w)(1-x)(1-y) \\ &= 1 - x - y + xy + wx + wy - 2wxy \\ &= 1 - x - y + xy + w(x + y - 2xy). \end{aligned}$$

For signed integers,

$$\begin{aligned} e &= (1-w)x + wx(1-y) + (1-w)(1-x)(1-y) \\ &= 1 - w - y + xy + wx + wy - 2wxy \\ &= 1 - w - y + xy + w(x + y - 2xy). \end{aligned}$$

The full protocol is given in Algorithm 25. There are three parallel executions of bitwise less-than protocol and then two multiplications. Sadly no work can be done during the offline phase, but the online phase has $b + 1 + 2 \cdot 1 = b + 3$ rounds. The communication cost is

$$3n(n-1)b(3b-1) + 2n(n-1)b = (9b-1)n(n-1)b \approx 9n(n-1)b^2.$$

If we want to use this algorithm with signed integers in the modified two's complement representation, we need to check if the values a and/or b are in the range $\{\lfloor \frac{p}{2} \rfloor + 1, ..., \lfloor \frac{p}{2} \rfloor + \lfloor \frac{d}{2} \rfloor\}$. To do that, we compare a and $a + \lfloor \frac{d}{2} \rfloor$ to half of the prime p. Table 6.6 shows how less-than can be calculated for that signed value representation, however, as this is not our chosen representation, we omit further details.

Algorithm 25: Less than LT($[\![a]\!], [\![b]\!]$)

```
Input: $[\![a]\!]$, $[\![b]\!]$
Result: $[\![e]\!]$, where $e = (a \overset{?}{<} b)$
1 $[\![w]\!] = \texttt{BLT}([\![a]\!], \frac{p}{2})$
2 $[\![x]\!] = \texttt{BLT}([\![b]\!], \frac{p}{2})$
3 $[\![y]\!] = \texttt{BLT}([\![a]\!] - [\![b]\!], \frac{p}{2})$
4 $[\![xy]\!] = [\![x]\!] \cdot [\![y]\!]$
5 $[\![x']\!] = [\![x]\!]$                    // (signed integers $[\![x']\!] = [\![w]\!]$)
6 $[\![e]\!] = 1 - [\![x']\!] - [\![y]\!] + [\![xy]\!] + [\![w]\!] \cdot ([\![x]\!] + [\![y]\!] - 2[\![xy]\!])$
```

a			b				
< 0	$[0, \frac{p}{2})$	$(\frac{p}{2}, \lfloor\frac{p}{2}\rfloor + \lfloor\frac{d}{2}\rfloor]$	< 0	$[0, \frac{p}{2})$	$(\frac{p}{2}, \lfloor\frac{p}{2}\rfloor + \lfloor\frac{d}{2}\rfloor]$	$a - b < \frac{p}{2}$	$a < b$
1	0	0	0	?	?	?	1
0	1	0	1	?	?	?	0
0	1	0	0	0	1	?	1
0	0	1	?	?	0	?	0
1	0	0	1	0	0		
0	1	0	0	1	0	0	1
0	0	1	0	0	1		
1	0	0	1	0	0		
0	1	0	0	1	0	1	0
0	0	1	0	0	1		

Table 6.6: Less-than for integers in modified two's complement representation

Theorem 35. The less-than without bit decomposition protocol in Algorithm 25 is correct.

Proof. To prove correctness, we need to show that e is one if $a < b$ and zero otherwise. First lets examine the situation, where either a or b is larger than half of the prime, but not both of them, i.e. $w \neq x$. Now, the result for unsigned integers is

$$\begin{aligned} e &= 1 - x - y + xy + w(x + y - 2xy) \\ &= \begin{cases} 1 - x - y + xy = 0 & \text{if } w = 0 \text{ and } x = 1 \quad \text{i.e. } a \in [\frac{p}{2}, p), b \in [0, \frac{p}{2}) \\ 1 - y + wy = 1 & \text{if } w = 1 \text{ and } x = 0 \quad \text{i.e. } a \in [0, \frac{p}{2}), b \in [\frac{p}{2}, p). \end{cases} \end{aligned}$$

The result for signed integers is

$$\begin{aligned} e &= 1 - w - y + xy + w(x + y - 2xy) \\ &= \begin{cases} 1 - y + xy = 1 & \text{if } w = 0 \text{ and } x = 1 \quad \text{i.e. } a \in (-\frac{p}{2}, 0), b \in [0, \frac{p}{2}) \\ 1 - w - y + wy = 0 & \text{if } w = 1 \text{ and } x = 0 \quad \text{i.e. } a \in [0, \frac{p}{2}), b \in (-\frac{p}{2}, 0). \end{cases} \end{aligned}$$

The alternative situation, i.e. both a and b are in the same region, hence $w = x$ gives us

$$\begin{aligned} e &= 1 - x' - y + xy + w(x + y - 2xy) \\ &= 1 - x' - y + x'y + x'(x' + y - 2x'y) \\ &= 1 - x' - y + x'y + x' + x'y - 2x'y \\ &= 1 - y \\ &= \begin{cases} 0 & \text{if } y = 1 \quad \text{i.e. } a - b \text{ did not overflow, hence } a \geq b \\ 1 & \text{if } y = 0 \quad \text{i.e. } a - b \text{ overflowed, hence } a < b. \end{cases} \end{aligned}$$

□

Theorem 36. The less-than without bit decomposition protocol in Algorithm 25 is secure against a passive adversary.

Proof sketch. All the sub-protocols are perfectly simulatable, hence this protocol is perfectly simulatable.

The resulting shares come from $[\![w]\!]$, $[\![x]\!]$, $[\![y]\!]$ and $[\![xy]\!]$, which were created by universally composable bitwise less-than and multiplication protocols. These shares, which are uniformly distributed and independent from the input shares $[\![a]\!]_t$ and $[\![b]\!]_t$ are then added or subtracted from each other. Therefore, similarly to Theorem 10, this protocol is universally composable. □

Chapter 7

Comparison of protection domains

7.1 Complexity

Table 7.1 summarises together all the complexities given in the previous sections of the protocols implemented during this work. For some of the algorithms, such as conjunction of bits and Prefix-AND, there are better solutions out there [DFK$^+$06, NO07]. Many of the latter algorithms, e.g. less-than for bitwise shared values, depend on them, see Figure 6.1. Therefore, future work improving the complexity of Prefix-Or also improves the complexities of equality testing and less-than comparison.

	Offline		**Online**	
Protocol name	Rounds	Data	Rounds	Data
`Classify`	0	0	1	nb
`Reshare`	1	$n(n-1)b$	0	0
`Declassify`	1	$k(k-1)b$	1	kb
$[\![a]\!] + [\![b]\!]$	0	0	0	0
$c[\![a]\!]$	0	0	0	0
$[\![a]\!] \cdot [\![b]\!]$	0	0	1	$n(n-1)b$
$\neg[\![b]\!]$	0	0	0	0
$[\![a]\!] \wedge [\![b]\!]$ $[\![a]\!] \vee [\![b]\!]$ $[\![a]\!] \oplus [\![b]\!]$	0	0	1	$n(n-1)b$
`Random`	1	$n(n-1)b$	0	0
`RandomBit`	2	$(2n+k)(n-1)b$	0	0

`Conjunct`
`Disjunct`
`PrefixAND`
`PrefixOR`
`PrefixXOR`
	0	0	$b-1$	$\approx n(n-1)b^2$

`BLT`	0	0	$b+1$	$\approx 3n(n-1)b^2$
`BitComposition`	0	0	0	0
`RandomBitwise`	$b+4$	$\approx (n-1)(5n+k)b^2$	0	0
`LSB`	$b+4$	$\approx (n-1)(5n+k)b^2$	$b+3$	$\approx 3n(n-1)b^2$
`LTHalfPrime`	$b+4$	$\approx (n-1)(5n+k)b^2$	$b+3$	$\approx 3n(n-1)b^2$
`EQPublic`	1	$n(n-1)b$	3	$(n+2k)(n-1)b$
`EQ`	$b+4$	$\approx (n-1)(5n+k)b^2$	b	$\approx (n-1)nb^2$
`LT`	0	0	$b+3$	$\approx 9n(n-1)b^2$

Table 7.1: Complexities for protocols in this work

	`additive3pp`				`shamirnpp`			
	Offline		**Online**		**Offline**		**Online**	
Protocol name	Rounds	Data	Rounds	Data	Rounds	Data	Rounds	Data
Classification	0	0	1	$3b$	0	0	1	$3b$
Resharing	1	$6b$	0	0	1	$6b$	0	0
Declassification	1	$6b$	1	$3b$	1	$2b$	1	$2b$
$[\![a]\!] + [\![b]\!]$	0	0	0	0	0	0	0	0
$c[\![a]\!]$	0	0	0	0	0	0	0	0
$[\![a]\!] \cdot [\![b]\!]$	1	$12b$	1	$3b$	0	0	1	$6b$
Equality	0	0	$\log_2 b + 2$	$22b + 6$	$b + 4$	$34b^2$	b	$6b^2 - 2b$
Less than	0	0	$\log_2 b + 3$	$12b \log_2 b + 48b + 16$	0	0	$b + 3$	$54b^2 - 6b$

Table 7.2: Complexities comparison

In order to better compare this work to the additive secret-sharing protocol suite, we created a PD for our `shamirnpp` PDK. As the additive scheme uses three parties, we also use three parties ($n = 3$), but as the multiplication protocol has a requirement $2k - 1 \leq n$, we need to make $k = 2$, hence 2-out-of-3 Shamir's scheme. We split the additive protocol suite complexities to online and offline phase. See Table 7.2 for comparison of rounds and data communication costs. One of the things to notice is that the multiplication and declassification protocols are theoretically better on `shamirnpp`. The more complicated operations, however, do not look that promising. We would like to stress, that this work is an initial attempt to implement a protocol suite inspired by Shamir's secret sharing scheme on Sharemind, while the protocol suite inspired by additive secret sharing has gotten many optimisations over the years.

7.2 Practical performance

To get the idea of how efficient our protocol implementations were, we benchmarked them against the additive three-party protocol suite. Here we use the same 2-out-of-3 Shamir's scheme as in complexities comparison. The initial benchmarking was done on different sizes of arrays of `uint8`'s. For the testing, we used a single laptop with 1.7 GHz processor for running all three $\mathcal{CP}$s. Multiplication comparison can be seen in Figure 7.1 and equality testing in Figure 7.2. The graphs additionally show how fast operations on public data were. The multiplication protocols offer similar performance, with our implementation being faster on smaller input sizes. The trend, however, is not in our favour and our multiplication is becoming slower as the input size increases. The Equality comparison graph, however, does not look that promising. In Figure 7.2, we can see that our protocol is about three times slower than the additive one.

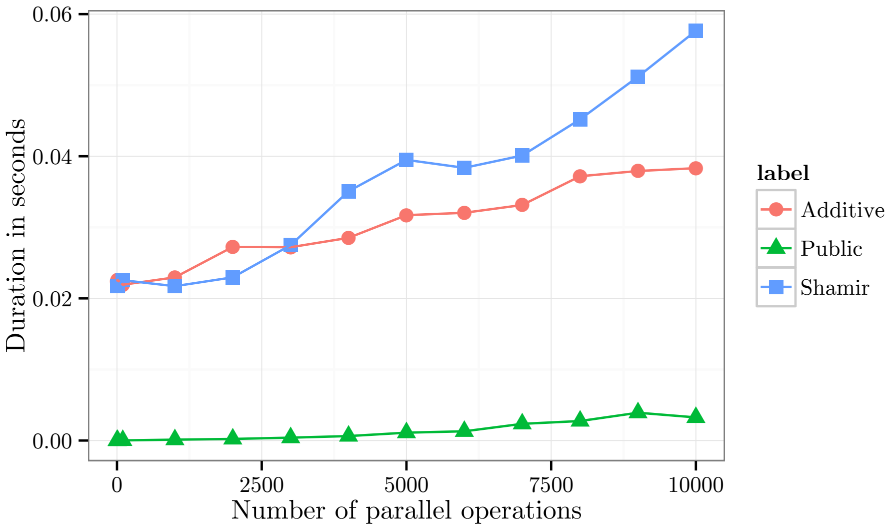

Figure 7.1: Multiplication performance comparison

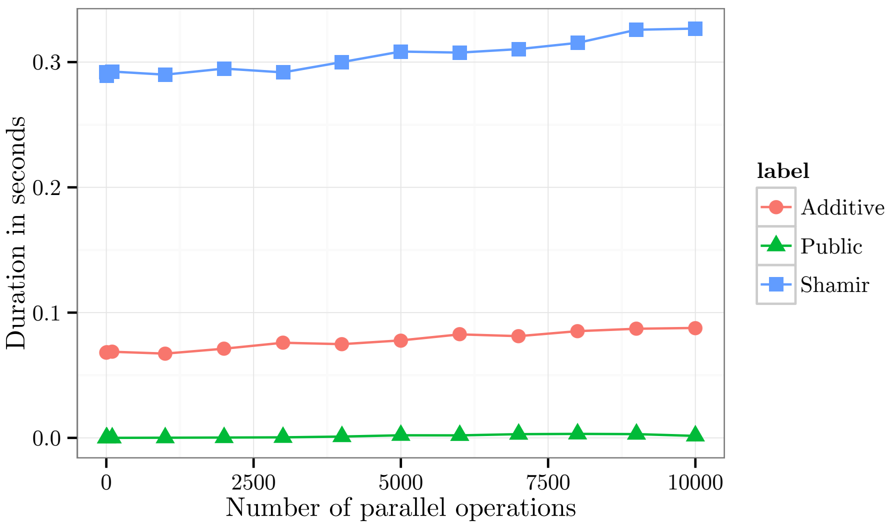

Figure 7.2: Equality performance comparison

Chapter 8 Conclusion

Secure multi-party computation allows us to perform analysis on private data without compromising it. Therefore, practical solutions for SMC are very welcome and Sharemind is one of the examples of such frameworks. There are already various protocol suites implemented on Sharemind, such as an additive three-party protocol suite. In this thesis, we designed and implemented a protocol suite, that was inspired by Shamir's secret sharing scheme. The latter is a popular way to divide a secret into pieces, called shares.

The main result of this thesis are the implemented protocols with correctness and security proofs. We created a new protection domain kind `shamirnpp`, that allows one to create protection domains for various n-out-of-k Sharmir's secret-sharing schemes. This PDK can now be used to write secure applications in the SecreC language. More specifically, we implemented protocols for addition, multiplication, boolean arithmetic and comparison operations. These protocols are the building blocks for various other functions one would want to possess, when analysing private data. As Sharemind has a standard library and a possibility to write domain-polymorphic code, many additional features, such as the absolute value function, can already be used with our newly implemented PDK.

The goal of this work was to explore another SMC implementation option and compare it to the existing one on Sharemind. Our new protection domain kind based on Shamir's scheme was compared to `additive3pp`. Looking at simpler protocols, such as declassification or multiplication, we saw that our SMC algorithms offer better theoretical complexity. That was also evident from the benchmarking results for smaller input sizes. For larger inputs and more complicated operations, such as equality testing and less-than comparison, we had to admit `additive3pp` being better. One of the reasons, for the performance difference, is our naive implementations for `Conjunct` and `PrefixAND` algorithms. Many other algorithms depend on their performance, see Figure 6.1, and improving it would improve the speed of equality testing and less-than comparison.

This brings us to future work. As mentioned before, some of the protocols from this thesis could be improved. There are also other algorithms that could be added to our protocol suite. For example, it may be useful, if we could convert shares into a different PD's shares. In this thesis, we in theory separated the offline and online phase, in practice, we did not. Shamir's k-out-of-n threshold scheme would allow to handle some $\mathcal{CP}$s disappearing or dealing with more corrupted parties. Exploring the implementation specifics of protocol interruption is an interesting topic for further research.

References

[ACS02] Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 417–432. Springer Berlin Heidelberg, 2002.

[AS11] Cybernetica AS. Deliverable D3.1: Technology-independent secure virtual machine architecture, 2011. Secure Virtual Machines and Languages–Project Technical document for Sharemind 3.

[BCD$^+$09] Peter Bogetoft, DanLund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, JanusDam Nielsen, JesperBuus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer Berlin Heidelberg, 2009.

[BDJ$^+$06] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Crescenzo and Avi Rubin, editors, *Financial Cryptography and Data Security*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer Berlin Heidelberg, 2006.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A System for Secure Multi-party Computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 257–266, New York, NY, USA, 2008. ACM.

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

[BLLP14] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14. IEEE Computer Society, 2014. To appear.

[BLR13] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-Polymorphic Programming of Privacy-Preserving Applications. In *Proceedings of the First ACM Workshop on Language Support for Privacy-enhancing Technologies, PETShop '13*, ACM Digital Library, pages 23–26. ACM, 2013.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206, Berlin, Heidelberg, 2008. Springer-Verlag.

[Bog13] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications.* PhD thesis, University of Tartu, 2013.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 15–15, Berkeley, CA, USA, 2010. USENIX Association.

[BTW12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis (short paper). In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security. FC'12*, pages 57–64, 2012.

[CD05] Ronald Cramer and Ivan Damgård. Multiparty computation, an introduction. In *Contemporary Cryptology*, Advanced Courses in Mathematics - CRM Barcelona, pages 41–87. Birkhäuser Basel, 2005.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *Theory of Cryptography*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer Berlin Heidelberg, 2005.

[CMF$^+$14] Koji Chida, Gembu Morohashi, Hitoshi Fuji, Fumihiko Magata, Akiko Fujimura, Koki Hamada, Dai Ikarashi, and Ryuichi Yamamoto. Implementation and evaluation of an efficient secure computation system using 'R' for healthcare statistics. *Journal of the American Medical Informatics Association*, 2014.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, JesperBuus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer Berlin Heidelberg, 2006.

[Flo63] Ivan Flores. *The Logic of Computer Arithmetic.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1963.

[Gei10] Martin Geisler. *Cryptographic Protocols:: Theory and Implementation.* PhD thesis, Aarhus University, Faculty of Science, Department, 2010.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 101–111, New York, NY, USA, 1998. ACM.

[Ham14] Koki Hamada. MEVAL: A Practically Efficient System for Secure Multi-party Statistical Analysis. Presented at Workshop on Applied Multi-Party Computation, Microsoft Research, Redmond, February 2014.

[HKS$^+$10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for Automating Secure Two-party Computations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 451–462, New York, NY, USA, 2010. ACM.

[Jag10] Roman Jagomägis. SecreC: a Privacy-Aware Programming Language with Applications in Data Mining. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[KBdH09] F. Kerschbaum, D. Biswas, and S. de Hoogh. Performance comparison of secure comparison protocols. In *Database and Expert Systems Application, 2009. DEXA '09. 20th International Workshop on*, pages 133–136, August 2009.

[KBLV13] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, 2013.

[KW13] Liina Kamm and Jan Willemson. Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. Cryptology ePrint Archive, Report 2013/850, 2013. http://eprint.iacr.org/.

[Mal11] Lior Malka. VMCrypt: Modular Software Architecture for Scalable Secure Computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 715–724, New York, NY, USA, 2011. ACM.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In *In USENIX Security Symposium*, pages 287–302, 2004.

[NN05] Ventzislav Nikov and Svetla Nikova. On proactive secret sharing schemes. In Helena Handschuh and M.Anwar Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 308–325. Springer Berlin Heidelberg, 2005.

[NO07] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography – PKC 2007*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360. Springer Berlin Heidelberg, 2007.

[Pul13] Pille Pullonen. Actively Secure Two-Party Computation: Efficient Beaver Triple Generation. Master's thesis, Institute of Computer Science, University of Tartu, 2013.

[Reb10] Reimo Rebane. An integrated development environment for the SecreC programming language. Bachelor's thesis. University of Tartu, 2010.

[Reb12] Reimo Rebane. A Feasibility Analysis of Secure Multiparty Computation Deployments. Master's thesis, Institute of Computer Science, University of Tartu, 2012.

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

[Tal11] Riivo Talviste. Deploying secure multiparty computation for joint data analysis—a case study. Master's thesis, Institute of Computer Science, University of Tartu, 2011.

[Vic61] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, 1961.

[WR67] E.T. Whittaker and G. Robinson. *The Calculus of Observations: A Treatise on Numerical Mathematics 4th ed.* Dover Publications, New York, 1967. §17 "Lagrange's Formula of Interpolation.".

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A General-purpose Compiler for Private Distributed Computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 813–826, New York, NY, USA, 2013. ACM.