UNIVERSITY OF TARTU

Institute of Computer Science
Computer Science Curriculum

Sander Siim

A Comprehensive Protocol Suite for Secure Two-Party Computation

Master's Thesis (30 ECTS)

Supervisor:	Dan Bogdanov, PhD
Supervisor:	Pille Pullonen, MSc

Tartu 2016

Terviklik protokollide kogu kahe osapoolega turvalisteks arvutusteks

Lühikokkuvõte:

Turvaline ühisarvutus võimaldab üksteist mitte usaldavatel osapooltel teha arvutusi tundlikel andmetel nii, et kellegi privaatsed andmed ei leki teistele osapooltele. Sharemind on kaua arenduses olnud turvalise ühisarvutuse platvorm, mis jagab tundlikke andmeid ühissalastuse abil kolme serveri vahel. Sharemindi kolme osapoolega protokolle on kasutatud suuremahuliste rakenduste loomisel. Igapäevaelus leidub rakendusi, mille puhul kahe osapoolega juurustusmudel on kolme osapoolega variandist sobivam majanduslikel või organisatoorsetel põhjustel. Selles töös kirjeldame ja teostame täieliku protokollistiku kahe osapoolega turvaliste arvutuste jaoks. Loodud protokollistiku eesmärk on pakkuda kolme osapoolega juurutusmudelile võrdväärne alternatiiv, mis on ka jõudluses võrreldaval tasemel. Kahe osapoole vahelised turvalise aritmeetika protokollid tuginevad peamiselt Beaveri kolmikute ette arvutamisele. Selleks, et saavutada vajalikku jõudlust, oleme välja töötanud tõhusad ette arvutamise meetodid, mis kasutavad uudsel viisil N-sõnumi pimeedastuse pikendamise protokolle. Meie meetodite eeliseks on alternatiividest väiksem võrgusuhtluse maht. Töös käsitleme ka insenertehnilisi väljakutseid, mis selliste meetodite teostamisel ette tulid. Töös esitame kirjeldatud konstruktsioonide turvalisuse ja korrektsuse tõestused. Selleks kasutame vähem eelduseid, kui tüüpilised teaduskirjanduses leiduvad tõestused. Üheks peamiseks saavutuseks on juhusliku oraakli mudeli vätimine. Meie kirjeldatud ja teostatud täisarvuaritmeetika ja andmetüüpide vaheliste teisendusprotokollide jõudlustulemused on võrreldavad kolme osapoole protokollide jõudlusega. Meie töö tulemusena saab Sharemindi platvormil teostada kahe osapoolega turvalisi ühisarvutusi.

Võtmesõnad: krüptograafia, turvaline ühisarvutus, kahe osapoole turvaline ühisarvutus, ühissalastus, Beaveri kolmikud, pimeedastus

CERCS: P170

A Comprehensive Protocol Suite for Secure Two-Party Computation

Abstract:

Secure multi-party computation allows a number of distrusting parties to collaborate in extracting new knowledge from their joint private data, without any party learning the other participants' secrets in the process. The efficient and mature Sharemind secure computation platform has relied on a three-party suite of protocols based on secret sharing for supporting large real-world applications. However, in some scenarios, a two-party model is a better fit when no natural third party is involved in the application. In this work, we design and implement a full protocol suite for two-party computations on Sharemind, providing an alternative and viable solution in such cases. We aim foremost for efficiency that is on par with the existing three-party protocols. To this end, we introduce more efficient techniques for the precomputation of Beaver triples using oblivious transfer extension, as the two-party protocols for arithmetic fundamentally rely on efficient triple generation. We reduce communication costs compared to existing methods by using 1-out-of-N oblivious transfer extension in a novel way, and provide insights into engineering challenges for efficiently implementing these methods. Furthermore, we show security of our constructions using strictly weaker assumptions than have been previously required by avoiding the random oracle model. We describe and implement a large amount of integer operations and data conversion protocols that are competitive with the existing three-party protocols, providing an overall solid foundation for two-party computations on Sharemind.

Keywords: cryptography, secure computation, two-party computation, secret sharing, Beaver triples, oblivious transfer extension

CERCS: P170

Acknowledgments

Foremost, I would like to thank my dear supervisors, from whom I have received tremendous support in bringing this work to a successful close.

The author along with the supervisors would like to thank Claudio Orlandi for his supportive ideas and helpful discussion on optimizing the precomputation phase.

The author of this thesis has received the Skype and Study IT in Estonia Master's Scholarship for the academic year 2015/16, funded by Estonian Information Technology Foundation and Skype Technologies OÜ.

Contents

1 Introduction

Secure multi-party computation (SMC) is a cryptographic method that allows mutually distrusting parties to perform computations on their joint data, while maintaining the privacy of their inputs. This seemingly paradoxical task of computing on data without access to it, was shown to be theoretically feasible in the seminal papers of the 80's [Yao82, GMW87, BGW88]. Since then, a large portion of cryptographic research has been dedicated to creating more efficient secure multi-party protocols for arbitrary computations with increasingly stronger security guarantees. As a result, many platforms that implement such protocols in order to build practical secure applications have emerged in the last 10 years using various secure computation techniques [ABPP15].

In this thesis, we focus on the Sharemind platform [Bog13]. Sharemind is one of the few SMC platforms that has been deployed in practical applications using real data [BTW12, BKK$^+$16]. In these applications, Sharemind uses very efficient protocols for ring arithmetic that are based on *additive secret sharing* [BNTW12, LR15]. In the multi-party setting (with at least three computing parties), these protocols provide information-theoretic security and are one of the most efficient methods for secure computations in the passive security model. However, they require at least three parties and assume an honest majority of participants. In some scenarios, finding three non-colluding organizations to participate in the computation can be a difficult task on its own.

Sharemind is designed and implemented in a modular way to support potentially many different cryptographic methods of ensuring the confidentiality of private data and performing privacy-preserving computations. As the main contribution of this thesis, *we design and implement a general-purpose state-of-the-art protocol suite for efficient secure two-party computation on Sharemind*, to complement the three-party protocol suite.

In the theoretical part of the work, we design a two-party protocol suite that supports efficient ring arithmetic and Boolean operations on secret-shared integers, providing a solid foundation for building secure computation applications in the two-party setting. The protocol suite is divided into two parts. First, we construct protocols for an *offline* precomputation phase that involves generating random secret-shared multiplication triples, also called *Beaver triples* [Bea91]. In the two-party setting, Beaver triples provide the best method for performing a multiplication on secret-shared integers.

Previously, a passively secure two-party computation on Sharemind has been developed, which uses the Paillier cryptosystem [Pai99] for Beaver triple generation [PBS12]. The computational cost for for this approach is however quite demanding. Recently, the implementation of [DSZ15] has shown the practicality of precomputation methods based on *oblivious transfer extension*, that achieve much better performance compared to homomorphic encryption techniques. Having this knowledge, we also explore oblivious transfer extension methods in our work.

We present efficient state-of-the-art protocols for oblivious transfer extension, and prove their security based on a formal framework specialized for secret sharing based secure computation [BLLP14]. Our approach allows to avoid relying on the *random oracle model*, which is required in a general setting for the oblivious transfer extension protocols of [ALSZ13] and [KK13]. Instead we rely on the concrete *correlation robustness* assumption for hash functions. We provide the main underpinnings of this proof without the random oracle in our work, illuminating the plausibility of this approach. Also, as the original paper for the [KK13] 1-out-of-N oblivious transfer extension protocol lacks formal justification of its security, we generalize the correlation robustness definition to this case and prove its security using this assumption. We argue that this definition can be thought of as a natural generalization to the original assumption.

In terms of our overall security model, we consider passive security in the strongest sense, achieving *universal composability* [Can01] due to the results of [BLLP14], which is a requirement for ensuring the same security guarantees as provided by Sharemind's three-party protocol suite. Specifically, universal composability ensures security is preserved in an arbitrary concurrent environment, for example, when the parties communicate over the public Internet.

Using the described oblivious transfer extension protocols we construct and implement efficient Beaver triple generation protocols. Especially, we describe a novel way to use 1-out-of-N oblivious transfer extension to reduce the overall communication compared to current state-of-the-art methods. The best comparison of our protocol suite can be made with the recent ABY framework for secure two-party computation, as it is based on a similar passively secure model and uses Beaver triple generation using oblivious transfer extensions [DSZ15]. We show our approach to reducing communication promises better overall performance. We discuss our implementation of the precomputation in detail and give insights to solving some of the engineering challenges we faced and venues of future optimization.

Relying upon Beaver triples and our implemented precomputation phase, we construct arithmetic computation protocols for the *online* phase, that allow manipulating secret-shared in a privacy-preserving manner. We give a detailed and comprehensive description of our online protocols, building on the line of work of optimizing arithmetic protocols for Sharemind's three-party protocol suite [BNTW12, LR15]. We adapt and optimize a number of efficient protocols for integer arithmetic, Boolean operations and data conversions into the two-party setting.

As the practical contribution of this work, we implement all described methods for both communication-efficient Beaver triple generation and the online arithmetic protocols on Sharemind, providing the foundation for building secure two-party computation applications. Our implemented oblivious transfer extension protocols allow us to easily extend the protocol suite with floating-point operations in the future, by combining it with a Yao's garbled circuits protocol implementation [PS15]. Compared to [DSZ15], we improve upon the communication complexity of the offline phase with our novel optimizations for oblivious transfer extension based Beaver triple generation, and also show a more efficient online phase. We also receive comparable performance to Sharemind's three-party protocol suite, showing that our work can be used for efficient real-world two-party applications in the future.

Contributions of the author. The author has researched the security and efficiency of existing best practices in the literature for Beaver triple precomputation in the passive model, including methods of oblivious transfer extension. The author has constructed and presented formal proofs of security for the oblivious transfer extension protocols and ultimately the Beaver triple generation protocol that uses them.

The author has implemented the described oblivious transfer extension protocols and the precomputation phase in C++ in a separate new protection domain on Sharemind. Especially, the author implemented novel constructions based on a 1-out-of-N oblivious transfer extension protocol. For all methods, the author has made significant efforts for optimizing the implementation of the precomputation by testing implementations of cryptographic primitives from different code libraries, leveraging special vectorized hardware instructions to build efficient bit matrix transposition algorithms and efficient pseudorandom number generation and using thread-parallelization techniques to parallelize hash function computations. The author has made and analyzed benchmarks of the implementation, comparing them to existing best results and providing insights into future optimization venues.

For the online phase of the protocol suite, the author has collected and summarized the most efficient protocols for integer arithmetic and conversions between secret sharing representations, based on best practices of Sharemind's three-party protocols. The author has studied both published papers and Sharemind's source code containing previously unpublished better optimized online arithmetic protocols. The author has understood how the protocols work, adapted them to the two-party setting, and provided their detailed formal descriptions. The described protocols were implemented by the author in C++ in the new two-party protection domain on Sharemind. The author also made benchmarks of the performance of these protocols and compared with existing efficient state-of-the-art implementations of two-party and three-party computation.

Thesis outline. We first begin in Section 2 with discussions and definitions of preliminary notions of secure multi-party computation that are required for presenting the protocols introduced in the rest of the work. Most notably, we outline our overall security framework in Section 2.5 that we use to prove the security of our constructions.

We move on to presenting protocols for oblivious transfer extension in Section 3, which play an integral role in the precomputation phase of our protocol suite. We also define the assumptions that allow us to show their correctness and privacy in our model, without using the random oracle.

In Section 4, we use the oblivious transfer extension protocols to construct secure protocols for Beaver triple generation. We describe our novel method of reducing communication, by using a 1-out-of-N oblivious transfer. We also discuss the implementation of our precomputation phase and present benchmarks for the described techniques.

Finally, in Section 5, we describe the overall structure of our protection domain and present formal descriptions of our implemented online arithmetic protocols in detail. We also present the benchmarks for the online phase and compare the performance of our implementation with existing state-of-the-art implementations.

2 Preliminaries

2.1 Notation

This section is meant as a reference of notation for the whole thesis and we already describe here notation for topics that are introduced later in the section. We aim for clarity and consistency with this notation throughout the work.

Sets and sampling Let S be a set. We denote by $x \leftarrow S$ sampling the element x uniformly randomly from S. We use the shorthand $[n] = \{1, \ldots, n\}$ for $n \in \mathbb{N}$ and use $\{0,1\}^n$ to denote the set of all bit strings with length n.

Functions We define functions with domain X and range Y as $f : X \to Y$. We write $f(\cdot)$ to denote that f is a function that takes a single argument or $f(\cdot, \ldots, \cdot)$ for multiple arguments.

Integers and modular arithmetic When using elements from a ring $\mathbb{Z}_N$, we implicitly assume all arithmetic on those values is done modulo N. That is, we omit the modulus from expressions such as $a = b + c \bmod N$, unless a different modulus is used.

For an integer $x \in \mathbb{Z}_{2^k}$, we use $x[i]$ to refer to the ith bit in its bitwise representation for $i \in [k]$. Then $x[1]$ refers to the least-significant bit. We also use $x[i..j]$ to refer to the array of consecutive bits from $x[i]$ up to and including $x[j]$ for $i < j$.

Vectors and matrices We denote vectors with bold letters, e.g $\mathbf{x} \in S^k$, where $\mathbf{x}$ is a vector of length k with elements from S. We denote the elements of $\mathbf{x}$ as $(x_1, x_2, \ldots, x_k)$ and we always start indexing vector elements from 1. Given $\mathbf{x} \in S^k$ and an index set $\mathcal{I} \subseteq [n]$, we use $\mathbf{x}_\mathcal{I}$ to denote a sub-vector of $\mathbf{x}$ containing only elements x_i for $i \in \mathcal{I}$. Thus, for $|\mathcal{I}| = m$, we have $\mathbf{x}_\mathcal{I} = (x_{i_1}, \ldots, x_{i_m})$, where $i_j \in \mathcal{I}$ and $i_k < i_l$ for all $k, l \in [n]$, such that $k < l$.

Similarly, we use bold capital letters to denote matrices. Let $\mathbf{A} \in S^{m \times n}$ be an $m \times n$ matrix. We denote the rows of $\mathbf{A}$ as vectors $\mathbf{a}_1, \ldots, \mathbf{a}_m$ with subscript indices and the columns as $\mathbf{a}^1, \ldots, \mathbf{a}^n$ with superscript indices.

Secret-shared values We use $[\![x]\!]$ to specify a value x that is secret-shared among a group of parties. The number of parties and their identities should always be clear from context. Let $[\![x]\!]$ be secret-shared among parties $\mathcal{P}_1$ and $\mathcal{P}_2$. Then, $[\![x]\!]_1$ and $[\![x]\!]_2$ stand for the individual shares held by $\mathcal{P}_1$ and $\mathcal{P}_2$ correspondingly, such that the original value x can be reconstructed from these shares. We naturally refer to vectors of secret-shared elements, as $[\![\mathbf{x}]\!] = ([\![x_1]\!], \ldots, [\![x_n]\!])$ for a vector $\mathbf{x}$ of length n. Note, that $[\![\mathbf{x}]\!]_1$ then refers to the shares of elements of $\mathbf{x}$, held by party $\mathcal{P}_1$ and e.g $[\![x_n]\!]_2$ refers to $\mathcal{P}_2$'s share of the n-th element of $\mathbf{x}$.

In most cases, the secret sharing scheme used for $[\![x]\!]$ should be clear from the context. For the additive secret sharing scheme, we sometimes refer specifically to a shared value over $\mathbb{Z}_N$ with $[\![x]\!]_{\bmod N}$. When we write $[\![x]\!] \in \mathbb{Z}_N$, we mean that the shares and also the combined value are from $\mathbb{Z}_N$. For bitwise sharing, we use $[\![x]\!]_{\oplus 2^\ell}$ to denote bitwise sharing over $\mathbb{Z}_2^\ell$.

When secret-shared variables are used in arithmetic formulae, they always refer to the *real combined* value of the variable. For example, writing $[\![x]\!] \odot [\![y]\!] = [\![z]\!]$ means that $z = x \odot y$ for some arithmetic operator $\odot$, and that all three of these values are secret-shared. When referring to a performed computation, $[\![x]\!] \odot [\![y]\!]$ means that a cryptographic protocol is carried out to compute the result, and this result is always obtained in the same secret-shared form. On the other hand, when we write $[\![x]\!]_1$, we explicitly refer to the value of the share held by $\mathcal{P}_1$ and not the combined value. Similarly, an expression such as $[\![x]\!]_1 \odot [\![y]\!]_1$ refers to a computation that can be carried out locally by $\mathcal{P}_1$, without communication with the other parties.

2.2 Secure multi-party computation

2.2.1 General problem description

The theory of secure multi-party computation (SMC) dates back over 30 years, and was first generalized and formulated by A. Yao in his 1982 seminal paper [Yao82]. The general setting is the following: a number of parties wish to collaboratively perform some computation on their private data without revealing this data to the other parties nor to anyone else. However, they still want to learn the result of the computation.

To be put more concretely, we have $n \geq 2$ parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, each with an input x_i from some fixed domain, who wish to compute a function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$. That is, each $\mathcal{P}_i$ inputs x_i and receives as output y_i. The informal security requirement is that no party should learn *anything else* except its prescribed output, although that output can depend on other parties' inputs. What is commonly referred to as *parties* in a secure computation can be considered as separate physical servers connected to a network, or more broadly as individuals or organizations each with their own computing resources and data.

Since the inception of SMC as one of the most fundamental research areas in cryptography, this rather broad problem description has been studied from a large variety of angles, employing various cryptographic techniques in different security models. The broad range and variety of published SMC research is exemplified in a recent paper by Perry et al., who propose a systematization of SMC methods, where techniques are classified along a total of 22 axes [PGFW14]. Stemming from the descriptions of theoretical protocols, a number of efficient SMC implementations have also appeared. A good overview and comparison of mature state-of-the-art SMC techniques and implementations can be found in [ABPP15]. Similarly, the number of real-life applications leveraging the work done in this vast research area has been steadily growing in the recent years. See for example [BCD$^+$09, BTW12, KBLV13, BJoSV15, Vah15, Sec, DDN$^+$15, BKK$^+$16].

2.2.2 Parties' roles in secure computation

In the literature, the standard treatment of SMC is that the parties involved in the secure computation themselves provide the private inputs to the computed function, and a subset of those parties also learn the result. For describing and showing security of protocols, this is enough, but for actual real-world applications, a more broader model should be considered. In a number of real-world use cases for SMC, different parties have different roles in the overall process. A more general model for SMC is presented in [BDNP08, BKL$^+$14], with three different roles for parties:

1. Input party $\mathcal{IP}$ — a data owner providing their confidential data as input to the computation.

2. Computing party $\mathcal{CP}_i$ — one of the parties who is actually involved in carrying out the secure computation protocols with input provided by input parties. Multi-party techniques require at least 2 computing parties.

3. Result party $\mathcal{RP}$ — a party to whom the computing parties disclose the result of the computation.

Note that any party can assume one or more of these roles at once in a given application. As an example, we can consider the canonical secure computation application from Yao's paper [Yao82]. In the millionaire's problem, two wealthy individuals would like to determine which one of them is richer, without disclosing the amount of capital they possess to the other party. In essence, the two parties, let us denote them $\mathcal{P}_1$ and $\mathcal{P}_2$, want to securely compute the less-than comparison function $f(x_1, x_2)$, which outputs $f(x_1, x_2) = 1$ if $x_1 < x_2$ and $f(x_1, x_2) = 0$ otherwise. Here, x_i is an integer representing the wealth of party $\mathcal{P}_i$. In this scenario, we have both $\mathcal{P}_1$ and $\mathcal{P}_2$ assuming all of the three roles described above. They give their input x_i compute the function $f(x_1, x_2)$ among themselves using a suitable SMC protocol and expect to receive the result.

The millionaire's problem is a good example for introducing the field of secure computation. However, SMC can also be applied to much more involved real-world use cases. One such example is a privacy-preserving government tax fraud detection system [BJoSV15].

In the application scenario presented in [BJoSV15], the government is predetermined to collect more detailed data about private companies' business transactions through their compulsory tax declarations, in order to detect tax fraud more efficiently. The government's motivation is to find tax-evading enterprises and thereby reduce the tax hole. The private sector could also benefit from such an arrangement, as the fraudulent companies with a dishonest competitive edge would be caught and eliminated from the market.

However, the honest companies would like to reduce the risk of their sensitive business secrets leaking to the competition, since a super-database of the country's trade and business network would be a prime target for an attacker outside or within the government. As such, a multi-party computation system would be a good solution to alleviate risks while benefitting all parties in the process.

The solution proposed in [BJoSV15] involves three computing parties with expected non-colluding interests, namely, the tax board, a representative organization of the private sector and a neutral government watch-dog organization. In this scenario, the input parties are the companies, who provide their data for tax fraud analysis. The companies themselves are involved in the actual computation indirectly through their representative organization, who takes the role of one concrete computing party. The tax board and neutral third organization also take the role of computing parties. However, only the tax board is allowed to receive the results of the fraud analysis, since making these completely public could impose undeserved reputational damage to some companies.

This scenario is a good example of the kinds of complex models that can be applied to secure multi-party computation, motivating the more general approach.

2.2.3 Security notions

We now give a high-level overview of the fundamental models of security used for SMC and how security is defined in these models. First, we limit the problem scope by noting some implicit restrictions we have already made in our previous discussion. One such restriction is that we assume the function f is known to all the computing parties, that is, we do not consider hiding the computed function itself. Similarly, we assume the length of the input arguments is also known by the parties. Although methods exist for hiding also the function and length of inputs, they are typically less efficient and build upon the same basic methods we discuss in this work [MS13, KS16, LMS16].

The basic properties that we do require from secure protocols are *privacy* and *correctness.* Let us fix a protocol π that computes the function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ between parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, where $\mathcal{P}_i$ inputs x_i and receives output y_i.

Informally, we say π *privately computes* f, if for any set of inputs $\{x_i\}_{i\in[n]}$, no party $\mathcal{P}_i$ learns anything else except which can be derived from x_i and y_i during the execution of π. As such, privacy assures the most intuitive notion of security, that the confidentiality of the inputs are preserved.

Notice however that a party can indirectly infer information about other parties' inputs from their own legal output. In this sense, there exist functionalities that might be privately computed by this definition, but can leak a lot of information about other parties inputs. This notion can be referred to as *output privacy* and is concerned with bounding the amount of information that the output itself gives about the inputs[1]. Output privacy is best viewed in the context of a specific application and real data owners and is out of the scope of this thesis[2].

Having a protocol that ensures privacy is enough to protect the confidentiality of inputs, but it does not guarantee that the protocol actually provides any meaningful results. As such, we require also *correctness* from π. Correctness simply requires for the output of the protocol π to align with the prescribed output of f on any set of inputs. If f is a randomized function, we require the output distributions to coincide. In the following, we refer to *secure* protocols as those that are both private and correct.

Having defined the intuitive properties we require from secure protocols, we next have to define the type of adversarial behavior against which these properties should be maintained. When arguing about security of cryptographic protocols, we consider an abstract entity called the *adversary*, that tries to break the security of the protocol. The adversary can corrupt a subset of the computing parties and force them to behave dishonestly, which models the malicious activity of a computing party in an actual execution of the protocol. There are two main types of adversarial corruption.

In the *passive model*, the corrupted party tries to infer extra information about the inputs of other parties from its view of the protocol execution, but follows the protocol description as defined. Such adversaries are called *semi-honest* or *honest-but-curious.*

The *active model* considers *malicious* adversaries that can arbitrarily deviate from the protocol description, for example, change their inputs, alter the contents

[1]Methods such as *differential privacy* allow to mathematically quantify the information leakage of different functions.

[2]Ultimately, it is up to the data owners to decide what they allow to do with their data. If data owners are themselves participating in the computation, they can simply choose not to engage in the computation, if they feel the chosen function leaks too much information. In other cases, output privacy might be indirectly enforced for example by appropriate legislation, ensuring that the process is properly audited.

of sent network messages or not send messages at all. As such, additional security properties have to be considered in the active model and achieving privacy and correctness is significantly more challenging in the malicious setting.

Additionally, we can consider different corruption strategies for the adversary. *Static* corruption assumes that the adversary chooses which parties to corrupt before the execution of the protocol. *Adaptive* corruption allows parties to be corrupted at any time during the protocol execution. In all cases, we should bound the number of parties that the adversary can corrupt, since no security mechanism can defend against an omnipotent adversary who corrupts all parties[3].

We can also consider some bounds on the computational capability of the adversary. In the *computational model* (providing *computational security*), we consider adversaries whose running-time is asymptotically bounded, typically by a polynomial of the security parameter. Computational security fundamentally rests on the assumed hardness of certain computational tasks. Security can be proven by a reduction showing, that in order to break the security of the protocol, the adversary would have to solve a computationally infeasible task. A common example is that of solving the discrete logarithm in some finite groups, on which a very large amount of cryptographic primitives rely.

In the *information-theoretic model* (providing *information-theoretic* or *perfect security*), there is no bound on the adversary's computational resources. This gives a stronger guarantee that no possible adversary can compromise security, regardless of their computational power. However, information-theoretic security can (in general) only be achieved in case of an honest majority of participants. Therefore, we rely on computational security for two-party computation.

For both models, the assumption of secure authenticated channels is made[4]. This means an adversary cannot read or change messages sent between computing parties without corrupting a computing party. We also adopt the *asynchronous* network model, allowing the adversary to have total control over network scheduling for messages sent between parties.

Lastly, the distinction of *stand-alone* or *concurrent* protocol execution can be made. Security in the stand-alone model gives guarantees only in the case where the communication links are dedicated for the computing parties and the specific protocol. The concurrent model makes no assumptions about other protocols that are being executed in the same network channels concurrently between the parties, and therefore gives better security guarantees in practice.

To summarize, in this work we consider passive static adversaries in a concur-

[3]In fact, security is meaningless in this scenario, since there are no honest parties left to protect.

[4]For the two-party case, in fact only authenticated channels are required. However, a secure channel can be easily constructed from an authenticated one by using symmetric cryptography, so we assume secure authenticated channels throughout the work for simplicity.

rent model with computational security.

2.3 Secret sharing

Secret sharing is a well-known cryptographic method for protecting secret data by distributing it amongst a group of parties. It was first proposed independently by both A. Shamir [Sha79] and G. Blakley [Bla79]. The original motivation for constructing secret sharing schemes in these seminal works was to protect the integrity and confidentiality of cryptographic keys. Later, secret sharing was used to construct efficient methods of general secure computation in the multi-party setting [BGW88, CCD88]. These papers form the basis for a line of work of many efficient SMC protocols that are based on evaluating arithmetic circuits, sometimes referred to as BGW-style protocols, quoting the Ben-Or, Goldwasser, Wigderson 1988 paper [BGW88].

The idea of secret sharing, is that a secret value x is broken into random-looking pieces $x_1, \dots, x_n$ called *shares*. The shares are then distributed among the involved parties, each getting a single share. The secret can later be constructed by combining any sufficiently large subset of the shares, however, smaller subsets give no information about the original value. We now formalize a secret sharing scheme in more concrete terms.

Definition 1 ((t,n)-secret sharing scheme). *A (t,n)-secret sharing scheme over $\mathcal{M}$ is defined by a randomized operation* $\mathsf{Share}(\cdot)$ *and an efficient deterministic operation* $\mathsf{Combine}(\cdot, \dots, \cdot)$ *with t arguments, with the following properties:*

- *For any value $x \in \mathcal{M}$,* $\mathsf{Share}(x)$ *outputs n shares $x_1, \dots, x_n \in \mathcal{M}$.*
- *Fix an arbitrary sharing of x, $(x_1, \dots, x_n) \leftarrow \mathsf{Share}(x)$. For any $\mathcal{I} \subseteq [n]$, such that $|\mathcal{I}| = t$, we have $\mathsf{Combine}(x_{i_1}, \dots, x_{i_t}) = x$, where $i_k \in \mathcal{I}$, $i_k \neq i_\ell$ for $k \neq \ell$.*

For simplicity, we assume in our definition that shares are chosen from the same domain $\mathcal{M}$ as the values that are secret-shared, as this is the case in many common schemes such as Shamir's secret sharing scheme [Sha79], and also the ones used in this work. The above definition covers one basic property of a secret sharing scheme, that it must always be possible to reconstruct the secret from the shares. However, it does not imply that no information can be learned about the secret from small subsets or even individual shares. We thus define next what it means for a secret sharing scheme to be *perfectly secure*.

Definition 2 (Perfect security of a secret sharing scheme). *We say a (t,n)-secret sharing scheme over $\mathcal{M}$ is perfectly secure iff for any $a, b \in \mathcal{M}$, indices $\mathcal{I} \subseteq [n]$,* *such that $|\mathcal{I}| < t$ and vector of shares $\hat{\mathbf{s}} \in \mathcal{M}^{|\mathcal{I}|}$, we have that*

$$\Pr[\mathbf{s}_\mathcal{I} = \hat{\mathbf{s}} \mid \mathbf{s} \leftarrow \mathsf{Share}(a)] = \Pr[\mathbf{s}_\mathcal{I} = \hat{\mathbf{s}} \mid \mathbf{s} \leftarrow \mathsf{Share}(b)] \ .$$

Intuitively, this definition says that any subset of the shares output by $\mathsf{Share}(\cdot)$ with less than t elements is equally likely to obtain any set of possible values, independently from the original secret. Consequently, without knowing at least t shares, no information can be learned about the secret. Note that this definition gives *information-theoretic* security, and therefore the strongest guarantee of privacy for secret sharing that we could hope for.

We now give two examples of secret sharing schemes that are perfectly secure according to our definition and which we use in our two-party protocol suite.

Definition 3. *An* additive secret sharing scheme *is an (n,n)-scheme over a ring $\mathbb{Z}_{2^k}$ for $k \in \mathbb{N}$ defined as follows.*

1. *For $x \in \mathbb{Z}_{2^k}$, $\mathsf{Share}(x)$ samples each share $x_1, \dots, x_{n-1}$ uniformly randomly from $\mathbb{Z}_{2^k}$, computes $x_n = x - \sum_{i=1}^{n-1} x_i$ and returns $(x_1, \dots, x_n)$.*
2. *$\mathsf{Combine}(x_1, \dots, x_n)$ computes $x = \sum_{i=1}^{n} x_i$ and returns x.*

It is easy to see that the additive scheme satisfies Def. 1, since

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n-1} x_i + x_n = \sum_{i=1}^{n-1} x_i + \left(x - \sum_{i=1}^{n-1} x_i \right) = x \ .$$

We can also show that the additive scheme is perfectly secure, using the following useful property of modular arithmetic.

Lemma 1. *For any random variable $X \in \mathbb{Z}_n$ and uniformly random $r \leftarrow \mathbb{Z}_n$, independent from X, we have that the random variable $X + r$ is also uniformly random and independent from X.*

Proof. For any $x, y \in \mathbb{Z}_n$, we have that

$$\Pr[X + r = y \mid X = x] = \Pr[r = y - x \mid X = x] = 1/n$$

since y, x are fixed values and r is uniformly random and chosen independently of x. Therefore, calculating the total probability we get

$$\begin{aligned} \Pr[X + r = y] &= \sum_{x \in \mathbb{Z}_n} \Pr[X + r = y \mid X = x] \cdot \Pr[X = x] \\ &= 1/n \sum_{x \in \mathbb{Z}_n} \Pr[X = x] = 1/n \ . \end{aligned}$$

Therefore, $X + r$ is distributed uniformly over $\mathbb{Z}_n$, independent of X. $\square$

Essentially, Lemma 1 says that for a random value $r \in \mathbb{Z}_n$, $x+r$ perfectly hides any value $x \in \mathbb{Z}_n$. In other words, $x + r$ is a one-time pad encryption of x. Note that the same properties hold for $x - r$. We now prove the security of the additive scheme.

Theorem 1 (Perfect security of additive secret sharing scheme)**.** *The (n,n)-additive secret sharing scheme described above is perfectly secure.*

Proof. Fix arbitrary $x \in \mathbb{Z}_{2^\ell}$ and a sharing $(x_1, \ldots, x_n) \leftarrow \mathsf{Share}(x)$. It is trivial to see that the condition in Def. 2 holds for subsets $\mathcal{I} \subseteq [n-1]$, since shares $x_1, \ldots, x_{n-1}$ are generated uniformly randomly and independently of x. Now consider a subset $\mathcal{I} \subset [n]$, such that $n \in \mathcal{I}$. There exists at least one index $j \in [n], j \neq n$, such that $j \notin \mathcal{I}$. We have that

$$x_n = x - \sum_{i \in [n], i \neq j} (x_i) + x_j \ .$$

Using Lemma 1, we have that x_n is a uniformly random value independent from all x_i, $i \in \mathcal{I}$, since x_j is uniformly random and independent from all x_i, $i \in \mathcal{I}$ and also x. Therefore all shares in the subset $\mathbf{s}_\mathcal{I}$ are uniformly random and independent from each other and x, for any $\mathcal{I} \subset [n]$. $\square$

Another very similar scheme is the bitwise secret sharing scheme.

Definition 4. *A* bitwise secret sharing scheme *is an (n,n)-scheme over a ring $\mathbb{Z}_2^k$ with bitwise XOR and conjunction operations, defined as follows.*

1. *For $x \in \mathbb{Z}_2^k$,* $\mathsf{Share}(x)$ *samples each share $x_1, \ldots, x_{n-1}$ uniformly randomly from $\mathbb{Z}_2^k$, computes $x_n = x \oplus \bigoplus_{i=1}^{n-1} x_i$ and returns $(x_1, \ldots, x_n)$.*

2. $\mathsf{Combine}(x_1, \ldots, x_n)$ *computes $x = \bigoplus_{i=1}^{n} x_i$ and returns x.*

The bitwise sharing scheme shares the same properties as the additive scheme. It has the property that $x \oplus r$ perfectly masks a value $x \in \mathbb{Z}_2^k$ for uniformly random $r \in \mathbb{Z}_2^k$ in terms of Lemma 1. Therefore, it is perfectly secure following the same reasoning as in Theorem 1. As the additive scheme works directly on integers from $\mathbb{Z}_{2^k}$, the bitwise scheme can be considered to work on the binary representation of integers. Both schemes are useful, as the additive scheme is more suitable for performing arithmetic operations and the bitwise scheme allows more efficient bit-level operations.

2.4 Sharemind secure computation platform

Sharemind is a secure computation platform that allows to build privacy-preserving applications in a general SMC model [Bog13]. Sharemind is designed in a modular way so support potentially different types of methods of secure computation, with varying amount of computing parties. A Sharemind module that provides a complete set of secure computation and private data storage tools is called a *protection domain* [BLR13].

The goal of a protection domain for Sharemind is to provide a complete set of primitives to allow the same kinds of computations that are done on regular data, but done in a privacy-preserving manner, relying on one or more specific SMC techniques. Primitives we might require are for example, standard arithmetic on integers and floating point numbers, different types of aggregation, filtering and statistical analysis. When we consider how these computations are done on a regular computer processor (CPU), we see that even the most complex analytical computations are eventually decomposed into a series of primitive instructions on the CPU. The instructions are scheduled in a specific order, and later computations may take as input partial results from previous steps.

Following this exact basic principle, we can build protocol suites for secure computation that allow practically arbitrary computations on private data from a rather small set of primitive protocols that perform small well-defined computational tasks. Using a suitable composition of these primitives, a variety of complex computations can be carried out. In theory, addition and multiplication is enough to construct any arithmetic circuit, but using separate fine-tuned protocols for the most often used primitives has proven to be a more efficient strategy for Sharemind's existing protocols [BNTW12].

Sharemind's most efficient protocols are in the three-party protection domain, that provides passive security against static adversaries that corrupt at most one of the parties [BNTW12, LR15]. Although the passive model provides less theoretical guarantees against more powerful adversaries, there are still many practical use cases where the passive model is sufficient. In many cases, the parties are honest in principle, but restricted from sharing their data with the other parties due to data protection laws, such as in [BKK$^+$16]. Additionally, protocols in the passive model are much more efficient and make it possible to build applications that process very large amounts of data in a privacy-preserving manner [BJoSV15].

The advantage of the three-party setting is that no precomputation phase as such is necessary at all, and the online phase is still very efficient. In this sense, the two-party case has a fundamental disadvantage, and is more complex to implement. However, the two-party model might be a better fit in some use cases. For example, the previously discussed tax fraud detection application of [BJoSV15] is in fact more naturally implemented as a two-party computation between the tax board and private companies' representative. This motivates the construction of an efficient two-party protection domain, even taking into account the added computational complexity, compared to three-party protocols.

On Sharemind, we can build sophisticated computations easily using a programming language SecreC [BLR13], which essentially allows to program a composition of primitive protocols in a C-like programming language. The SecreC language was designed to be domain-polymorphic from its inception. This means that high-level algorithms programmed in SecreC for the existing three-party protection domain can be easily switched to use our built two-party protection domain instead, demonstrating the benefits of this design choice.

2.5 Security and composition of SMC protocols

We now discuss the security framework for Sharemind's BGW-style protocols, that is, protocols that process secret-shared data between $n \geq 2$ parties. In the context of this thesis, we only consider the passively secure case with static adversaries. Considering Sharemind's protection domain architecture, it is clear that the security framework should allow to prove privacy and correctness for individual primitive protocols. Additionally, and more importantly, we have to ensure that this notion of security is preserved when the protocols are executed in *arbitrary composition* with each other. Also, other protocols may be running in parallel on the same communication channels. For example, if the protocol is executed over the public Internet, we can have arbitrary other protocols running concurrently with ours.

Protocols that achieve security under general composition in a concurrent environment are called *universally composable* following the definitions of the universal composability (UC) framework of Canetti [Can01, CCL15]. For Sharemind protocols, a similar and slightly specialized framework was formalized in [BLLP14], based on the reactive simulatability (RSIM) framework [PW00]. The RSIM and UC frameworks are technically different formalizations for arguing about security of protocols under composition, however in our case, for a fixed amount of parties, they give similar guarantees [Can00, DKMR05]. The exact details are out of the scope of this thesis.

When considering only Sharemind's protocols based on secret sharing, the general UC framework is overly restrictive and the framework of [BLLP14] allows for more efficient protocols, while maintaining the strong security under general composition. An important distinction is that we consider privacy as the main goal for proving security of our protocols, since we can later get full security easily from a simple composition. For our purposes, we only give a high-level description of the [BLLP14] framework with enough detail to argue about security for the protocols presented in this work. For the more rigorous technical details, we refer the reader to [BLLP14].

2.5.1 Modeling protocol execution

In the RSIM model, a computing party is modeled as a *state transition machine* M with input and output *ports* for communicating with other machines, a set of *states*, with specific initial and final states and a probabilistic *state transition function*, which defines the machine's behavior. Data transfer between machines is performed through *buffers*, which connect an input port of one machine with the output port of another. When a machine writes a value to a buffer, the receiving end machine is clocked, which means it is executed and uses the state transition function to determine its next state. As a result, the machine may write one or many outputs to some of its output ports.

Each computing party is modeled as a separate machine, with communication channels between them. The parties receive their inputs from a machine called the *environment* H. The environment models all possible external parties and other protocols that may be running concurrently on the same communication channels as a single abstract entity. The environment has designated buffers for communicating with each of the computing parties. In a secure computation, the environment gives the inputs to the computing parties, thereby invoking their execution and reads the outputs they eventually write to their output ports. During the protocol execution the machines may send a number of messages to each other, before writing the final output.

Adversarial behavior is modeled by another separate machine called the *adversary* $\mathcal{A}$. The adversary also has buffers for communication with each computing party, and can send a special corrupt message to any computing party machine. Upon receiving a corrupt message, the machine writes its entire protocol view to the buffer connected to $\mathcal{A}$, consisting of received inputs and written outputs up to this point. Afterwards, the corrupted party immediately sends each new received input and written output also to $\mathcal{A}$. Additionally, the adversary and the environment have a free two-way communication channel, which essentially means that the adversary potentially shares its entire view also to the environment. Also, when we are considering an execution of a system of machines, any non-determinism in the clocking order of the machines is determined by the adversary.

Since we are modeling passive corruption, the outputs written by the computing parties are not influenced by corrupt messages. We also assume secure, authenticated channels, meaning that the adversary cannot read or change messages sent by uncorrupted parties. However, we allow the adversary to control the timing of the entire protocol network scheduling, including receiving inputs and sending outputs to the environment. Since we use static corruption, we assume the corruption messages are sent before the protocol execution starts[5].

2.5.2 Privacy and security definitions

The security definitions of [BLLP14] follow the *real vs ideal world* paradigm. In this approach, the security of a protocol is defined in terms of an ideal functionality, that acts as a trusted third party, and is without flaw in terms of security. We then contrast this ideal scenario with an actual implementation of the protocol. The goal for a security proof is to show, that the real world execution of the protocol is indistinguishable from the ideal execution, from the viewpoint of the adversary. If this is the case, then we can reason, that the adversary cannot achieve its malicious goals any better in the real world than it could have in the ideal world, as he cannot himself distinguish between these scenarios.

The security guarantee from this kind of approach is one of the strongest, sometimes referred to as *fully-simulatable* security. Simulatability refers to the fact that showing equivalence of the real and ideal worlds requires us to construct a *simulator* machine, which can simulate the real world view to an adversary that is placed in the ideal world. If this construction is achievable, then by a rather intricate deduction, we can say that the adversary could have simulated the real world for himself, without relying on any information that it sees in the real world. This in turn proves that the real world view does not leak anything interesting to the adversary.

For the definition of security, the real execution of an n-party protocol is modeled as previously described, where the computing parties are represented as machines $\hat{M} = (M_1, \ldots, M_n)$ along with their connecting buffers. Additionally, the environment H and adversary $\mathcal{A}$ have buffers to communicate with each M_i and also with each other. We call this set-up of a system of machines $\hat{M}$, H and $\mathcal{A}$ as a *configuration* $(\hat{M}, H, \mathcal{A})$.

We define security through the indistinguishability of the environment's view compared to an ideal execution of the protocol. We should now specify how we define ideal functionalities. Note that it is paramount to define the ideal functionality correctly, as this is the reference standard to which we compare the security of our protocols. In most cases, the ideal functionality is modeled a single machine that performs the required computation as a trusted third party. For an ideal functionality that securely computes the function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, the machine $\mathcal{F}$ simply takes all the inputs $(x_1, \ldots, x_n)$, computes f on these inputs and outputs $(y_1, \ldots, y_n)$.

To model the ideal execution, we replace the parties $\hat{M}$ by the machine $\mathcal{F}$, that takes all parties' inputs from the environment H, performs the required computa-

[5]Note that the composition results from [BLLP14] also hold for adaptive corruption.

tions, and writes the outputs back to the environment. Note that $\mathcal{F}$ still responds to the adversary's $\mathsf{corrupt(i)}$ messages by sending the input and output of the i-th party to the adversary. However, as $\mathcal{F}$ is a single monolithic machine, there are no intermediary protocol messages to send. As such, the simulator should be able to simulate these messages to the adversary. The simulation must ensure that the whole simulated view is then consistent with the real inputs and outputs seen by the environment, that is, the entire view of H. This must hold also for any environment and adversary. For a given configuration $\mathsf{conf} = (\hat{M}, H, \mathcal{A})$, we define $\mathsf{view}_{\mathsf{conf}}(H)$ as the random variable which consists of the ordered inputs and outputs as seen by H in the protocol execution concatenated together.

To give the security definition we must first define a notion of computational indistinguishability for probability distributions. A function $\mu(\cdot)$ is negligible if for every positive polynomial $p(\cdot)$ and all sufficiently large n, it holds that $\mu(n) < 1/p(n)$.

Definition 5 (Computational indistinguishability)**.** *We say that two probability distributions $\mathcal{D}_X$ and $\mathcal{D}_Y$ are computationally indistinguishable, denoted by $\mathcal{D}_X \overset{c}{\approx} \mathcal{D}_Y$, if for every non-uniform polynomial-time algorithm $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$ such that,*

$$|\Pr[\mathcal{A}(\mathcal{D}_X) = 1] - \Pr[\mathcal{A}(\mathcal{D}_Y) = 1]| \leq \mu(n) \ .$$

In the above definition, we mean by $\mathcal{A}(\mathcal{D})$ that the distinguisher is given access to an oracle that efficiently samples elements from $\mathcal{D}$. We now give our definition of security, which we use for the protocols presented in this work.

Definition 6 (Security)**.** *Let π be a protocol executed by n parties, modeled by the collection of machines $\hat{M} = (M_1, \ldots, M_n)$ and let $\mathcal{F}$ be an ideal functionality. We say that π is computationally secure w.r.t $\mathcal{F}$, iff for every configuration $\mathsf{conf}_1 = (\hat{M}, H, \mathcal{A})$, there exists a configuration $\mathsf{conf}_2 = (\mathcal{F}, H, \mathcal{S}_{\mathcal{A}})$ with the same H such that the views of the environment H are computationally indistinguishable*

$$\mathsf{view}_{\mathsf{conf}_1}(H) \overset{c}{\approx} \mathsf{view}_{\mathsf{conf}_2}(H) \ .$$

Note that the view of H can arbitrarily depend on the view of $\mathcal{A}$ and even contain the entire view, since they share a communication channel. To show security, we have to construct a simulator $\mathcal{S}_{\mathcal{A}}$ that can simulate a corrupted party's messages sent during the protocol execution based only on the input and output of that party. We write $\mathcal{S}_{\mathcal{A}}$ to denote black-box simulation using $\mathcal{A}$. The simulated view should be indistinguishable from a normal execution of the protocol from the environment's perspective (which indirectly includes $\mathcal{A}$), meaning that the simulated messages also have to lead to the same output distribution that is defined by the ideal functionality.

For input privacy, we use a slightly different configuration of machines. The difference with full security is that we do not require the simulation outputs to coincide with the real outputs, but we still show correctness separately based on the protocol description. As such, we divide the environment H into two machines H_{in} and H_{out}. H_{in} gives inputs to the parties (or the ideal functionality) and communicates with $\mathcal{A}$, whereas the outputs are given only to H_{out} and are not forwarded to any other machine. Then, we only require that the view of H_{in} remains indistinguishable in the simulation, that is, the inputs and intermediary protocol messages, but excluding the outputs. We call this set-up $(\hat{M}, H_{in} \cup H_{out}, \mathcal{A})$ a *privacy configuration.*

Definition 7 (Input privacy). *Let π be a protocol executed by n parties, modeled by a collection of machines $\hat{M} = (M_1, \ldots, M_n)$ and let $\mathcal{F}$ be an ideal functionality. We say that π is computationally input-private w.r.t $\mathcal{F}$, iff for every privacy configuration $\mathsf{conf}_1 = (\hat{M}, H_{in} \cup H_{out}, \mathcal{A})$ there exists a privacy configuration $\mathsf{conf}_2 = (\mathcal{F}, H_{in} \cup H_{out}, \mathcal{S})$ with the same $H_{in} \cup H_{out}$, such that the views of the restricted environment H_{in} are computationally indistinguishable*

$$\mathsf{view}_{\mathsf{conf}_1}(H_{in}) \overset{c}{\approx} \mathsf{view}_{\mathsf{conf}_2}(H_{in}) \ .$$

For the simulation, we can only rely on the corrupted party's input for showing input privacy, since the output is not accessible from H_{out}. However, we also do not require the simulated view to lead to the same outputs that the ideal functionality computes, which makes the simulation easier.

Note that for both definitions, we can also consider the information-theoretic version, which requires the distributions of the real and simulated views of the environment to exactly match. We say *perfectly secure* and *perfectly input-private* to refer to the information-theoretic definitions.

The input privacy property as defined is clearly weaker than full security. The rationale for considering this separately, is that for protocols that take secret-shared inputs and produce secret-shared outputs, we can use input privacy to guarantee that nothing is learned about the other party's input share. However, the difference with full security is that later, if we publish both result shares to a party, the values of the individual shares seen together might leak more than intended, since they are not guaranteed to be independent of the input shares. However, we can explicitly make sure to re-randomize the shares before publishing them, to ultimately guarantee security. In general, input-private protocols are more efficient than secure protocols, which makes this approach advantageous in terms of performance.

From the composition theorems of [BLLP14], we get two main results:

1. The composition of input-private protocols remains input-private.

2. The composition of an input-private and secure protocol results in a secure protocol, if the input-private protocol correctly implements the ideal functionality.

For our case with protocols based on secret sharing, the first result allows us to build an arithmetic circuit based on input-private protocols as long as the final output shares are not directly published. Note that this holds for a strict composition, where all inputs to the protocol are the direct outputs of other input-private protocols. To publish the final result, we can use the second composition theorem and add a secure resharing step to the end of the circuit, to make the whole composition secure.

Note that there is an additional requirement for composition of input-private secure protocols to remain secure, namely we require the input-private protocol to correctly implement its ideal functionality in an honest execution. For randomized functionalities, this requires the output to be of the correct random distribution for a distinguisher that only sees the inputs to the protocol. For protocols that operate on secret-shared values, we require that the combined secret-shared result is exactly correct with respect to the arithmetic operation. For the proofs of security for composition, we refer the reader to [BLLP14].

Overall, the results allow us to essentially base our whole protocol suite on input-private protocols. We discuss these topics more in Section 5 when presenting our full protocol suite.

Note also, that these composition results hold in a more general concurrent setting, as we have made no restrictions to what the environment does. Modeling the environment this way, similarly to the UC framework, captures all other protocols that may be running in the network, including independent instances of the same protocol.

2.5.3 Ideal functionality for Sharemind protocols

For all protocols of our implemented two-party protection domain, we assume an ideal functionality, which is specific to protocols that operate on secret-shared values. The protocols follow the pattern of taking one or more secret-shared values as input and outputting a single secret-shared result. If the values for the inputs are $x_1, \ldots, x_n$, then the protocol computes some function $f(x_1, \ldots, x_n) = y$. We define the corresponding ideal functionality $\mathcal{F}$ as follows. $\mathcal{F}$ takes both parties' shares of $[\![x_1]\!], \ldots, [\![x_n]\!]$ as inputs. Then, $\mathcal{F}$ combines the shares to learn the real values and computes $y = f(x_1, \ldots, x_n)$.

A crucial step here is that $\mathcal{F}$ should output a fresh uniformly random sharing of y, that is, explicitly call $([\![y]\!]_1, [\![y]\!]_2) \leftarrow \mathsf{Share}(y)$. Otherwise, a valid sharing would be for example $(y, 0)$, which immediately reveals the result of the computation to the first party. The security property we seek, of course, is that the computing parties would not actually learn the result of the computation, but only a random sharing of it. Thus, the output shares of a secure protocol should be uniformly random and independent from the input shares. We denote such ideal functionalities simply by $\mathcal{F}(\llbracket x_1 \rrbracket, \ldots, \llbracket x_n \rrbracket) = \llbracket y \rrbracket$.

The correctness of such functionalities is defined by the requirement $y = f(x_1, \ldots, x_n)$, that is, the combined secret-shared output should give the right value with respect to the function f. Showing input privacy does not imply correctness, which is why we must show correctness separately. Clearly, it is easy to construct input-private protocols that simply produce arbitrary results, however, such protocols are obviously not very useful. Full security according to our definition also implies correctness by itself.

2.6 Oblivious transfer

We now turn to one of the most fundamental protocols for secure computation, called *oblivious transfer.* Although a conceptually simple primitive, oblivious transfer (OT) is a basic building block for a number of more interesting and useful cryptographic tasks. A famous result from J. Kilian [Kil88] shows that the existence of a secure oblivious transfer protocol is sufficient to securely evaluate any computable function between two parties. Additionally, commitment schemes and non-interactive zero-knowledge proofs can be constructed from oblivious transfer.

The notion of oblivious transfer was first proposed by M. Rabin in 1981 [Rab81, Rab05] (the digital version was published later) and then formalized to the current standard definition of 1-out-of-2 OT in [EGL85], which we describe now.

OT is a protocol between two parties, a *sender* $\mathcal{S}$ and a *receiver* $\mathcal{R}$. In its simplest variant of 1-out-of-2 OT (which we denote $\binom{2}{1}$-OT), $\mathcal{S}$ has two bit strings m_0, m_1 and $\mathcal{R}$ has a choice bit $b \in \{0, 1\}$. Given these inputs, the OT protocol outputs m_b to $\mathcal{R}$ according to its input choice bit, and the sender learns nothing. The informal security requirements for OT are that

1. the sender $\mathcal{S}$ does not learn which of the messages was chosen, maintaining *privacy of $\mathcal{R}$'s input*
2. and that the receiver $\mathcal{R}$ learns nothing about the other message m_{1-b}, preserving *partial privacy of $\mathcal{S}$'s input.*

There also exist many other natural generalizations to this basic definition. In $\binom{n}{1}$-OT, the receiver chooses one message out of n, and in $\binom{n}{k}$-OT, the receiver chooses a subset of size k from among n messages. An even more general variant exists, called *reactive* $\binom{n}{k}$-OT, where the receiver chooses a total of k messages, but the index for each next message is chosen after receiving the previous message, allowing the previous messages to influence the decision of the receiver. In this thesis, we are concerned only with the standard $\binom{n}{1}$-OT as a building block for our protocols.

We have already stated that OT can be used for constructing secure computation protocols between two parties. A concrete instantiation of this is the GMW protocol [GMW87]. The seminal paper by Goldreich-Micali-Wigderson proves a completeness theorem for secure multi-party computation in the case where a majority of participants is honest. The paper shows that $\binom{2}{1}$-OT is enough to evaluate any function securely (even with malicious adversaries).

Oblivious transfer is also used as an irreplaceable subroutine in the well-known two-party general computation technique called Yao's garbled circuits protocol [Yao82, LP09]. As such, OT is a well-studied protocol in the literature, since having a more efficient OT protocol also makes other protocols relying on it more practical.

Formally, the $\binom{2}{1}$-OT_ℓ ideal functionality computes $\mathcal{F}((x_0, x_1), b) = (\perp, x_b)$, where $x_i \in \{0,1\}^\ell$ and $b \in \{0,1\}$. We denote by $\perp$ that the sender gets no output from the protocol. More generally, the $\binom{n}{1}$-OT_ℓ functionality is defined by $\mathcal{F}((x_1, \ldots, x_n), r) = (\perp, x_r)$, where $x_i \in \{0,1\}^\ell$ and $r \in [n]$.

Note that for oblivious transfer, our input privacy definition is contradictory, since the protocol is defined to reveal one of the inputs of the sender to the receiver. However, as we discuss in the next section, we can consider variants of the standard oblivious transfer functionality, where the messages are not fixed by the sender before the protocol execution. In this case, also input privacy is meaningful.

3 Oblivious transfer extension

In this section, we describe protocols for efficiently performing oblivious transfer, which is the main building-block for our Beaver triple generation protocol, presented in Section 4.

Oblivious transfer is used in many secure computation protocols, as we have discussed in Section 2.6. In these protocols, usually many separate invocations of $\binom{n}{1}\text{-}\mathsf{OT}$ are required on different inputs[6]. However, oblivious transfer is a rather costly primitive since all known OT protocols require some form of computation-intensive public-key operations (for example [NP01, Lin08, CO15]). To be able to perform a large amount of OT-s efficiently, a technique called OT extension can be used [IKNP03]. OT extension protocols allow extending a small number of *base* OT invocations on random inputs to a much larger number of OT invocations for the actual inputs. The advantage is that the actual OT-s are much cheaper to perform, by using only symmetric-key cryptographic operations. Since the base OT-s are performed on random inputs, they are independent of the inputs used for the extended OT-s, which makes it possible to perform these as a precomputation.

For Section 4, we require a protocol to compute m invocations of $\binom{n}{1}\text{-}\mathsf{OT}$ in parallel, on m separate sets of inputs. We denote this functionality as $\binom{n}{1}\text{-}\mathsf{OT}^m_\ell$, meaning m parallel invocations of $\binom{n}{1}\text{-}\mathsf{OT}$ on ℓ-bit messages. In fact, we utilize variants of the standard OT functionality in our protocols, where the sender's inputs are actually not fixed, but rather generated as a result of the protocol. These protocols allow us to use the input privacy property, which is not applicable for standard OT.

In this work, we consider the passively secure protocols from [ALSZ13] and [KK13][7]. The ALSZ13 protocol implements $\binom{2}{1}\text{-}\mathsf{OT}$ and the KK13 protocol implements a more general $\binom{n}{1}\text{-}\mathsf{OT}$. In fact, the ALSZ13 protocol is a special case of the KK13 protocol for $n = 2$, but we analyze them separately for a clearer exposition.

We present protocols for *correlated* and *random* OT extension, that are later used in Section 4 for constructing secure Beaver triple generation protocols. Especially, we use $\binom{n}{1}\text{-}\mathsf{OT}$ to construct novel precomputation protocols with reduced communication compared to using $\binom{2}{1}\text{-}\mathsf{OT}$ for multiplication triples.

In the original papers of [ALSZ13, KK13], the security for these protocols is shown in the *random oracle model* [BR93]. A security proof in the random oracle model only gives a heuristic argument for security, as random oracles (uniformly randomly chosen functions) cannot exist in the real world [CGH04]. Also, for the KK13 protocol, a formal security proof has so far not appeared in the literature to our knowledge, but it's security is based on similar considerations as that of the

[6]For example, in Yao's protocol, a $\binom{2}{1}\text{-}\mathsf{OT}$ is performed for each input bit.

[7]There also exist OT extension protocols for the malicious case, for example [ALSZ15, KOS15].

ALSZ13 protocol.

In this work, we take a step towards using these protocols in a secure manner while *not relying* on the random oracle model. By this we mean, that we can construct fully secure protocols for Beaver triple generation, while only requiring correctness and input privacy from the oblivious transfer extension protocols. Our insight is that we can show input privacy and correctness by only relying on the *correlation robustness* property of a hash function [IKNP03]. For the KK13 protocol, we have to generalize this property to give a formal proof of input privacy. In all cases we assume a secure protocol exists to implement the base OT. Note that using only the privacy property is possible in our setting with secret sharing protocols, but may not applicable in other settings.

The proofs of input privacy that we present should be seen as convincing arguments that the random oracle model is not needed in our case. However, more rigorous analysis is required to show convincingly that the composition result of [BLLP14] holds for the input private OT protocols, with regard to composing them with a secure protocol. We note that the correctness requirement for the pseudorandom distribution of the outputs (for an external distinguisher) is important for this.

Arguably, the correlation robustness property is also a fairly strong assumption to make if we implement the hash with a real-world function, such as SHA-256 in our implementation. Still, this is a concrete advancement in the theoretical sense and allows also to show security in a concurrent model, which is otherwise difficult, if not impossible to do rigorously, when programmable random oracles are required[8].

With the exception of security in the concurrent settings, assuming the random oracle model would immediately imply security of our constructions following the proofs in [ALSZ13]. However, we specifically try to avoid the random oracle and take a first best-effort step in that direction in this work.

3.1 Correlated OT extension

In certain applications, a more restricted OT functionality than the standard $\binom{n}{1}$-OT can be used to build more efficient protocols. In *correlated OT*, instead of specifying two arbitrary messages $m_0, m_1 \in \mathcal{M}$, the sender fixes a correlation function $f : \mathcal{M} \to \mathcal{M}$, where $\mathcal{M}$ is the message space. As the result of the protocol, the sender receives a pseudorandom m_0 and the transferred messages are m_0, m_1, where $m_1 = f(m_0)$. In this section, we present the correlated OT extension

[8]To show UC security, we need to assume the random oracle is local to the specific protocol execution. However, a random oracle implemented as a real-life hash function is public to everyone, which contradicts the UC requirements [CJS14].

protocol from [ALSZ13], which is the most efficient passively secure protocol of its kind currently known.

For our purposes, we consider specific correlation functions of the form $f(x) = c + x$ for $\mathcal{M} = \{0,1\}^\ell$, where $c \in \mathcal{M}$ is some fixed *correlation offset* and addition is modulo 2^ℓ. We can then define the ideal functionality of $\binom{2}{1}$-COT as $\mathcal{F}_{\binom{2}{1}\text{-}\mathsf{COT}}(c, b) = (s, s + bc)$, where s is computationally indistinguishable from a uniformly random value from $\mathcal{M}$, independent of the inputs. The receiver then either learns s or $s + c$, depending on the choice bit b. We can thus consider input privacy for correlated OT, since the receiver should not learn the correlation offset c. Our proofs of correctness and input privacy rely on the correlation robustness property, which first appeared in [IKNP03] and is also used to show security of the standard $\binom{2}{1}$-OT extension protocol of [ALSZ13].

Definition 8 (Correlation robustness). *An efficiently computable function $H : \{0,1\}^\kappa \to \{0,1\}^\ell$ is said to be correlation-robust for m messages, if*

$$(t_1, \ldots, t_m, H(t_1 \oplus s), \ldots, H(t_m \oplus s)) \overset{c}{\approx} U_{m\cdot(\kappa+\ell)}$$

for uniformly random and independent choices of $t_1, \ldots, t_m, s$ from $\{0,1\}^\kappa$, where $U_{m\cdot(\kappa+\ell)}$ denotes the uniform distribution over $\{0,1\}^{m\cdot(\kappa+\ell)}$.

Note that the definition assumes that the distinguisher does not have access to the value of s directly. Also, note that a uniformly random function (a random oracle) trivially has this property. Therefore, this is strictly a weaker assumption than assuming the random oracle model for an m polynomial in κ.

Based on this definition, we prove in Lemma 2 that the output of a correlation-robust function on uniformly random inputs is uniformly random. We use this lemma to ascertain that the message chosen in the correlated OT protocol is pseudorandom.

Lemma 2. *Let $H : \{0,1\}^\kappa \to \{0,1\}^\ell$ be a correlation-robust function for m messages. Then, for random and independent choices of $t_1, \ldots, t_m \in \{0,1\}^\kappa$ we have that*

$$(H(t_1), \ldots, H(t_m)) \overset{c}{\approx} U_{m\cdot\ell}\ .$$

Proof. We can prove the lemma using a simple reduction against the correlation robustness of H. Assume a distinguisher $\mathcal{D}$ with non-negligible advantage in distinguishing $(H(t_1), \ldots, H(t_m))$ from the uniform distribution. Then we can construct a distinguisher $\mathcal{D}_{CR}$ against the correlation robustness of H. On input $(t_1, \ldots, t_m, H(t_1 \oplus s), \ldots, H(t_m \oplus s))$, $\mathcal{D}_{CR}$ obtains $b \leftarrow \mathcal{D}(H(t_1 \oplus s), \ldots, H(t_m \oplus s))$ and outputs b. Then, $\mathcal{D}_{CR}$ has exactly the same advantage in breaking correlation robustness of H, since $(t_1 \oplus s, \ldots, t_m \oplus s)$ are uniformly random and independent values. Therefore, this reduction is tight, with no loss in security parameters as $\mathcal{D}_{CR}$ performs no extra computations. $\square$

We also note that if a function $H : \{0,1\}^\kappa \to \{0,1\}^\ell$ is correlation-robust, then we can easily construct a correlation-robust function $H' : \{0,1\}^\kappa \to \{0,1\}^t$, where $t < \ell$, simply using the t first bits of the output of H. The reduction is trivial, since if a distinguisher $\mathcal{D}_{H'}$ can break the correlation robustness of H', it can certainly do so for H, by only considering the t first bits of the output of H and using the output of $\mathcal{D}_{H'}$ on these bits.

Additionally, we give a definition of a *pseudorandom generator* (PRG).

Definition 9 (Pseudorandom generator)**.** *We say an efficiently computable function $G : \{0,1\}^\kappa \to \{0,1\}^m$ is a secure pseudorandom generator, if*

$$G(k) \overset{c}{\approx} U_m$$

for a uniformly random $k \leftarrow \{0,1\}^\kappa$, where U_m denotes the uniform distribution over $\{0,1\}^m$.

Here also, we assume the distinguisher does not have direct access to the value of k. Therefore, a pseudorandom generator produces a stream of values that are computationally indistinguishable from uniform randomness, given a uniformly random seed k.

We now have all cryptographic tools that we need to present the correlated version of [ALSZ13] OT extension protocol as Protocol 1.

Theorem 2. *The ALSZ13 correlated OT extension protocol in Protocol 1 is correct, assuming a correlation-robust hash function H, secure PRG G and a computationally secure protocol for $\binom{2}{1}$-$\mathsf{OT}^\kappa_\kappa$..*

Proof. For the bit matrix $\mathbf{Q}$ constructed by $\mathcal{S}$, we have that the j-th row equals

$$\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j \ .$$

If $r_j = 0$, then $\mathbf{q}_j = \mathbf{t}_j$ and at the end of the protocol, the receiver $\mathcal{R}$ computes

$$H(\mathbf{t}_j) = x_j^0 \ .$$

For $r_j = 1$, $\mathbf{q}_j = \mathbf{s} \oplus \mathbf{t}_j$ and $\mathcal{R}$ computes

$$\begin{aligned} y_j \oplus H(\mathbf{t}_j) &= x_j^1 \oplus H(\mathbf{q}_j \oplus \mathbf{s}) \oplus H(\mathbf{t}_j) \\ &= x_j^1 \oplus H(\mathbf{t}_j) \oplus H(\mathbf{t}_j) = x_j^1 = x_j^0 + c_j \ . \end{aligned}$$

Therefore, the receiver $\mathcal{R}$ learns the correct message $x_j^{r_j} = x_j^0 + r_j \cdot c_j$ for each $j \in [m]$. We additionally have to show that x_j^0 is pseudorandom. This follows from Lemma 2, since the values of $\mathbf{q}_j$ are computationally indistinguishable from uniform randomness, due to security of $\binom{2}{1}$-$\mathsf{OT}^\kappa_\kappa$ and the PRG G. $\square$

Protocol 1 Correlated OT extension from [ALSZ13]

```
Functionality: $\binom{2}{1}$-$\mathsf{COT}_\ell^m$
Setup: Security parameter $\kappa$, correlation robust hash function $H : \{0,1\}^\kappa \to \{0,1\}^\ell$, PRG $G : \{0,1\}^\kappa \to \{0,1\}^m$
Input: $\mathcal{S}$ has $m$ correlation offsets $\mathbf{c} = (c_1, \ldots, c_m)$, $c_j \in \{0,1\}^\ell$. $\mathcal{R}$ has $m$ choice bits $\mathbf{r} = (r_1, \ldots, r_m) \in \{0,1\}^m$ .
Result: $\mathcal{S}$ gets pseudorandom $(x_1^0, \ldots, x_m^0)$ and $\mathcal{R}$ gets $(x_1^0 + r_1 \cdot c_1, \ldots, x_m^0 + r_m \cdot c_m)$
    Bootstrap phase
 1: $\mathcal{S}$ generates random bit string $\mathbf{s} = (s_1, \ldots, s_\kappa)$
 2: $\mathcal{R}$ generates random $\kappa$-bit seed pairs $(k_i^0, k_i^1) \in \{0,1\}^{2\kappa}$ for $i \in [\kappa]$
 3: Perform $\binom{2}{1}$-$\mathsf{OT}_\kappa^\kappa$ with choices $s_i$ and messages $k_i^0, k_i^1$
    Online phase
 4: $\mathcal{R}$ generates a $m \times \kappa$ bit matrix $\mathbf{T}$ with columns $\mathbf{t}^i = G(k_i^0)$ and rows $\mathbf{t}_j$
 5: $\mathcal{R}$ computes $\mathbf{u}^i = \mathbf{t}^i \oplus G(k_i^1) \oplus \mathbf{r}$ and sends $\mathbf{u}^i$ to $\mathcal{S}$ for each $i \in [\kappa]$
 6: $\mathcal{S}$ computes $\mathbf{q}^i = (s_i \cdot \mathbf{u}^i) \oplus G(k_i^{s_i})$        $\triangleright$ $\mathbf{q}^i = (s_i \cdot \mathbf{r}) \oplus \mathbf{t}^i$
 7: $\mathcal{S}$ builds a $m \times \kappa$ bit matrix $\mathbf{Q}$ with columns $\mathbf{q}^i$ and rows $\mathbf{q}_j$ $\triangleright$ $\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j$
 8: $\mathcal{S}$ computes $x_j^0 = H(\mathbf{q}_j)$ and $x_j^1 = x_j^0 + c_j$ for $j \in [m]$
 9: $\mathcal{S}$ sends $y_j = x_j^1 \oplus H(\mathbf{q}_j \oplus \mathbf{s})$ to $\mathcal{R}$ for $j \in [m]$
10: $\mathcal{R}$ computes $x_j^{r_j} = (r_j \cdot y_j) \oplus H(\mathbf{t}_j)$ for $j \in [m]$
11: return $(x_j^0, x_j^{r_j})$ for all $j \in [m]$
```

The correlated OT extension is more efficient then the protocol for standard $\binom{2}{1}$-OT presented in [ALSZ13], since the latter requires sending two messages y_j^0 and y_j^1 in the online phase, whereas the correlated version sends only one message. Also, not all m transfers have to be done at the same time, rather, they can be streamlined, since each transfer only requires calculating the next row in the matrices $\mathbf{T}$ and $\mathbf{Q}$. Since $\mathcal{R}$ generates the bit matrix $\mathbf{T}$ column-wise, but needs to access its rows to compute $H(\mathbf{t}_j)$, a bit matrix transposition is needed in the implementation. Similarly, also $\mathcal{S}$ has to transpose the matrix $\mathbf{Q}$. This can be a computational bottleneck, as shown originally in [ALSZ13], meaning an efficient transposition algorithm is important for performance.

Theoretically, the matrix $\mathbf{T}$ can also be precomputed entirely by the receiver, but this does not give much practical performance gains, since generating pseudorandomness is not the bottleneck of the protocol, but rather, bit matrix transposition and hashing are.

We now prove the input privacy of the correlated OT extension protocol.

Theorem 3. *The ALSZ13 correlated OT extension protocol in Protocol 1 is computationally input-private, assuming a secure PRG G, a correlation-robust hash function H and a computationally secure protocol for $\binom{2}{1}$-$\mathsf{OT}_\kappa^\kappa$.*

Proof. We show a simulation for both corrupted sender and receiver in the hybrid model where the parties have access to an ideal $\binom{2}{1}\text{-}\mathsf{OT}^{\kappa}_{\kappa}$ functionality.

Corrupted sender. In the hybrid model, the corrupted sender sends $\mathbf{s}$ to the ideal $\binom{2}{1}\text{-}\mathsf{OT}^{\kappa}_{\kappa}$ functionality. In the simulation, we forward this to $\binom{2}{1}\text{-}\mathsf{OT}^{\kappa}_{\kappa}$ and also simulate the receiver's inputs k_i^0, k_i^1 for $i \in [\kappa]$ by generating uniformly random values and give them as inputs to $\binom{2}{1}\text{-}\mathsf{OT}^{\kappa}_{\kappa}$. The simulator then receives $k_i^{s_i}$ for $i \in [\kappa]$ from $\binom{2}{1}\text{-}\mathsf{OT}^{\kappa}_{\kappa}$ and sends it to the corrupted receiver. Overall, this simulation of the bootstrap phase is perfect, since k_i^0, k_i^1 are also generated uniformly randomly in the real protocol by the receiver.

For the online phase, the simulator simply chooses uniformly random $\mathbf{u}^i$ for each $i \in [\kappa]$. Since the sender has no information about $k_i^{1-s_i}$ (due to security of $\binom{2}{1}\text{-}\mathsf{OT}$), we have that the value computed in the real protocol $\mathbf{u}^i = G(k_i^0) \oplus G(k_i^1) \oplus \mathbf{r}$ is computationally indistinguishable from the uniform distribution, as it is masked by $G(k_i^{1-s_i})$. Therefore, this simulation is perfect.

Corrupted receiver. To simulate the bootstrapping phase for corrupted receiver, the simulator simply receives k_i^0, k_i^1 from corrupted receiver. Since the receiver does not get any output from the bootstrapping phase, this is enough to simulate it.

In the online phase, we are left with having to simulate the message y_j. The simulator first generates the values of $\mathbf{t}^i$ and $\mathbf{u}^i$ as in the protocol description, based on the received k_i^0, k_i^1. We then consider the cases of $r_j = 0$ and $r_j = 1$ separately (the simulator can make the distinction since it knows the receiver's input $\mathbf{r}$). In the real protocol, we have that

$$y_j = (x_j^0 + c_j) \oplus H(\mathbf{q}_j \oplus \mathbf{s}) = (x_j^0 + c_j) \oplus H((r_j \cdot \mathbf{s} \oplus \mathbf{t}_j) \oplus \mathbf{s}) \ .$$

For $r_j = 0$, we have that this equals $(x_j^0 + c_j) \oplus H(\mathbf{t}_j \oplus \mathbf{s})$. The receiver knows the values of $\mathbf{t}_j$, but not $\mathbf{s}$. Therefore, due to the correlation robustness property, we have that $H(\mathbf{t}_j \oplus \mathbf{s})$ is computationally indistinguishable from a uniform distribution for all $j \in [m]$. Note that computational indistinguishability holds, since $\mathbf{t}_j$ is generated by a secure pseudorandom generator. Therefore, any advantage gained by the distinguisher in the correlation robustness definition due to $\mathbf{t}_j$ being pseudorandom not uniformly random, can be reduced to an attack against the pseudorandom generator G[9]. This also means, that y_j is computationally indistinguishable from uniform randomness and we can simulate it by generating a uniformly random value.

[9]We admit that there is a downgrade from the level of security of the PRG and correlation-robust hash for the whole protocol due to this reduction, which requires a more formal analysis.

For $r_j = 1$, we have that $y_j = (x_j^0 + c_j) \oplus H(\mathbf{t}_j)$ in the real world. In this case, we simulate y_j first by generating uniformly random x_j^1 and computing the value $x_j^1 \oplus H(\mathbf{t}_j)$. If $r_j = 1$, then $x_j^0 = H(\mathbf{q}_j) = H(\mathbf{t}_j \oplus \mathbf{s})$. Using the same argument as for the case $r_j = 0$, we have that x_j^0 is indistinguishable from uniform randomness, and therefore, so is $x_j^1 = x_j^0 + c_j$ due to Lemma 1 in Section 2.3, which matches our simulation. $\square$

Note that the proof applies actually to all cases where the correlation function f is a bijection on the message space, since then we have that $f(x_j^0)$ is uniformly random for uniformly random x_j^0. We also stress that the input privacy property together with the correctness guarantee of the output distributions is enough to show security of the Beaver triple generation protocol in Section 4 through composition with a secure protocol. Intuitively, input privacy gives us the guarantee that the outputs of both parties are independent of the other party's input. Correctness ensures that an honest execution produces the correct output distributions in the sense that the sender's output x_j^0 is pseudorandom and the receiver gets $x_j^0 + c_j$.

In addition, we point out that from Protocol 1 implementing $\binom{2}{1}\text{-}\mathsf{COT}_\ell^m$, we can trivially construct a protocol for $\binom{2}{1}\text{-}\mathsf{COT}_t^m$, where $t < \ell$ that retains all the security properties we have shown. We simply replace the correlation-robust hash H with H' that outputs the first t bits of the output of H, which retains correlation robustness. Naturally, this holds also, if H is modeled as a random oracle.

3.2 Random OT extension

Another useful variant of oblivious transfer that we use in our protocol suite is *random* OT. In this case, both of the sender's messages are arbitrarily generated in the protocol, without even a correlation. As such, the sender actually has no input to this protocol. We can then define the ideal functionality $\binom{2}{1}\text{-}\mathsf{ROT}$ as $\mathcal{F}_{\binom{2}{1}\text{-}\mathsf{ROT}}(\bot, b) = ((s_0, s_1), s_b)$, where both s_0 and s_1 are computationally indistinguishable from uniform randomness.

The random OT extension protocol from [ALSZ13] reduces communication even more, by not requiring to send the message y_j in the online phase at all. We present the protocol as Protocol 2. The protocol is nearly identical to the correlated OT version, except the message x_j^1 is also calculated directly from the hash output.

However, we need to assume a slightly different notion of correlation robustness in this case, to show that the values x_0^j, x_1^j are pseudorandom.

Definition 10 (Output correlation robustness)**.** *An efficiently computable function $H : \{0,1\}^\kappa \to \{0,1\}^\ell$ is said to be output correlation-robust for m messages, if*

$$(H(t_1), \dots, H(t_m), H(t_1 \oplus s), \dots, H(t_m \oplus s)) \overset{c}{\approx} U_{m \cdot 2\ell}$$

for uniformly random and independent choices of $t_1, \dots, t_m, s$ *from* $\{0,1\}^\kappa$.

As with the original correlation robustness property, we have that a uniformly random function is output correlation robust, similarly to the original correlation robustness definition. However, there is no trivial reduction to the original correlation robustness property, and as such, it is not clear whether this is a stronger or weaker assumption to make. We have currently left this analysis out of the scope of the thesis.

Protocol 2 Random OT extension from [ALSZ13]

Functionality: $\binom{2}{1}\text{-}\mathsf{ROT}^m_\ell$
Setup: Security parameter κ, output correlation robust hash function $H : \{0,1\}^\kappa \to \{0,1\}^\ell$, PRG $G : \{0,1\}^\kappa \to \{0,1\}^m$
Input: $\mathcal{R}$ has m choice bits $\mathbf{r} = (r_1, \dots, r_m) \in \{0,1\}^m$
Result: $\mathcal{R}$ gets $(x_1^{r_1}, \dots, x_m^{r_m})$, $\mathcal{S}$ gets pseudorandom x_j^0, x_j^1 for $j \in [m]$

```
    Bootstrap phase
 1: $\mathcal{S}$ generates random bit string $\mathbf{s} = (s_1, \dots, s_\kappa)$
 2: $\mathcal{R}$ generates random $\kappa$-bit seed pairs $(k_i^0, k_i^1) \in \{0,1\}^{2\kappa}$ for $i \in [\kappa]$
 3: Perform $\binom{2}{1}\text{-}\mathsf{OT}^\kappa_\kappa$ with choices $s_i$ and messages $k_i^0, k_i^1$
    Online phase
 4: $\mathcal{R}$ generates a $m \times \kappa$ bit matrix $\mathbf{T}$ with columns $\mathbf{t}^i = G(k_i^0)$ and rows $\mathbf{t}_j$
 5: $\mathcal{R}$ computes $\mathbf{u}^i = \mathbf{t}^i \oplus G(k_i^1) \oplus \mathbf{r}$ and sends $\mathbf{u}^i$ to $\mathcal{S}$ for each $i \in [\kappa]$
 6: $\mathcal{S}$ computes $\mathbf{q}^i = (s_i \cdot \mathbf{u}^i) \oplus G(k_i^{s_i})$        $\triangleright$ $\mathbf{q}^i = (s_i \cdot \mathbf{r}) \oplus \mathbf{t}^i$
 7: $\mathcal{S}$ builds a $m \times \kappa$ bit matrix $\mathbf{Q}$ with columns $\mathbf{q}^i$ and rows $\mathbf{q}_j$ $\triangleright$ $\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j$
 8: $\mathcal{S}$ computes $x_j^0 = H(\mathbf{q}_j)$ and $x_j^1 = H(\mathbf{q}_j \oplus \mathbf{s})$ for $j \in [m]$
 9: $\mathcal{R}$ computes $x_j^{r_j} = H(\mathbf{t}_j)$ for $j \in [m]$
10: return $((x_j^0, x_j^1), x_j^{r_j})$ for $j \in [m]$
```

Theorem 4. *The ALSZ13 random OT extension protocol in Protocol 2 is correct, assuming an output correlation-robust hash function H, secure PRG G and secure $\binom{2}{1}\text{-}\mathsf{OT}^\kappa_\kappa$ protocol.*

Proof. Since $\mathbf{q}_j = \mathbf{t}_j \oplus (r_j \oplus \mathbf{s})$, it is easy to see that $H(\mathbf{t}_j) = x_j^{r_j}$. In addition we have that to show that x_j^0 and x_j^1 are pseudorandom. This follows directly from the output correlation robustness definition Def. 10, since the values of $\mathbf{q}_j$ are computationally indistinguishable from uniform randomness and $\mathbf{s}$ is generated uniformly randomly. □

Note that in the case random OT, the sender has no input and as such, having correctness and input privacy alone does not guarantee that the receiver learns no information about the other message generated by the sender. However, this follows directly from the output correlation robustness assumption.

Theorem 5. *The ALSZ13 random OT extension protocol in Protocol 2 is computationally input-private, assuming a secure PRG G and a computationally secure protocol for $\binom{2}{1}$-$\mathsf{OT}^{\kappa}_{\kappa}$. The messages $x_j^{1-r_j}$ are computationally indistinguishable from uniform randomness to the receiver, for receiver's choice bits $r_j \in \{0,1\}$ for $j \in [m]$ assuming H is a output correlation robust function.*

Proof. For a corrupted sender, the simulation is identical to the correlated OT protocol simulation in Theorem 3. For corrupted receiver, we only have to simulate messages in the bootstrap phase, which is also identical to the first part of the proof regarding corrupted receiver in Theorem 3. For the simulation, we do not use the output correlation robustness.

However, to show that $\mathcal{R}$ cannot learn the messages $x_j^{1-r_j}$, we directly invoke the output correlation robustness definition, since then, $x_j^{1-r_j}$ is pseudorandom independently of $x_j^{r_j}$ for all $j \in [m]$. □

3.3 1-out-of-N OT extension

We now present the the [KK13] protocol for $\binom{n}{1}$-COT. Again we present the correlated version directly, as it is more communication-efficient than the corresponding $\binom{n}{1}$-OT protocol. Interestingly, the KK13 protocol is equivalent to the ALSZ13 protocol for the case $n = 2$, although the papers appeared independently.

As the original paper does not present any formal proof of security or required security assumptions [KK13], we present our own proof, which is naturally derived from the ALSZ13 security proof. Our first task is to generalize the correlation robustness property for the $\binom{n}{1}$-OT case, which we present in Def. 11.

Definition 11. *An efficiently computable function $H : \{0,1\}^\kappa \to \{0,1\}^\ell$ is said to be n-correlation-robust for m messages and $c_1, \ldots, c_n \in \{0,1\}^\kappa$ where $n < \kappa$, if it holds that*

$$\begin{aligned}(t_1, \ldots, t_m,\ & H(t_1 \oplus (s \odot c_{r_1,1})), \ldots, H(t_1 \oplus (s \odot c_{r_1,n})),\\ & \ldots,\\ & H(t_m \oplus (s \odot c_{r_m,1})), \ldots, H(t_m \oplus (s \odot c_{r_m,n})))\\ & \overset{c}{\approx} U_{m\cdot(\kappa+\ell(n-1))}\end{aligned}$$

for uniformly random and independent choices of $t_1, \ldots, t_m, s$ *from* $\{0,1\}^\kappa$ *and fixed values* $c_1, \ldots, c_n$*, where* $c_{i,j} = c_i \oplus c_j$*. Each row* $H(t_i \oplus (s \odot c_{r_i,1})), \ldots, H(t_i \oplus (s \odot c_{r_i,n}))$ *contains all elements except for the element* $H(t_i \oplus (s \odot c_{r_i,r_i})$*, therefore a total of* $n-1$ *elements. The values* $1 \leq r_i \leq n$ *for* $i \in [m]$ *are chosen by the distinguisher.*

Here we denote the point-wise product of two bit strings with $\odot$. We now explain the reasoning behind this definition. First notice that for $n = 2$, if we choose $c_1 = 0^\kappa$ and $c_2 = 1^\kappa$ (all zeroes and all ones), the definition is exactly equivalent to correlation robustness as defined in Def. 8. We also mention that the n-correlation robustness as defined seems to be a special case of a more general *correlated input pseudorandomness* definition presented in [GOR11], but the exact relationships between the definitions requires more analysis.

In our case for the $\binom{n}{1}$-COT protocol, we require that the values c_i should have a pair-wise minimum distance as high as possible. That is, the value $d = \min_{i,j\in[n],i\neq j} c_i \oplus c_j$ should be high, so that the security parameters are not affected as much. Since we are interested in the minimum distance of these elements, we can refer to them as codewords. We now give some informal arguments as to how the value of n affects the security parameter κ in the definition. Intuitively, it is clear that as n is larger, the distinguisher sees more hash outputs and can make a more informed decision and we therefore need to increase the security parameter.

If we consider the 2-correlation robustness definition Def. 8, then we can argue that, irrespective of the hash function used, the distinguisher has a better chance of distinguishing the hash outputs from randomness in two cases. First, if there are collisions in the inputs of the hashes, there are guaranteed repeating values also in the outputs. Secondly, if the secret value s happens to be zero, then the distinguisher easily wins, as he knows the inputs to the hashes and can compute the hash on these and compare it to the challenge.

In Def. 8 specifically, the collision of inputs is actually not a problem, since this happens exactly with the same probability as when uniformly random values are given to the distinguisher instead of the hash outputs. The probability of $s = 0$ is exactly $2^{-\kappa}$, so overall, we can postulate that the definition gives us κ-bit security, given that the hash outputs are "sufficiently random".

In the general n-correlation robustness definition, there are more possibilities for the value $s \odot c_{r_{i,j}}$ to be 0. Here is precisely where we consider the minimum distance d of the codewords $c_1, \ldots, c_n$, since $\Pr[s \odot c_{i,j} = 0] \leq 2^{-d}$ for fixed i, j. Since there are $\binom{n}{2}$ elements $c_{i,j}$ with $i \neq j$, we have that the total probability for generating s such that any $s \odot c_{i,j} = 0$ is bounded by $\binom{n}{2} \cdot 2^{-d}$.

However, we also have to consider collisions in the hash inputs, that is, the probability of having $t_i \oplus (s \odot c_{r_i,v}) = t_j \oplus (s \odot c_{r_j,u})$. In case of $i = j$, we certainly have $u \neq j$ and the collision happens exactly if $s \odot (c_{r_i,v} \oplus c_{r_j,u}) = s \odot c_{u,v} = 0$, for which we know the probability is 2^{-d}. Given that there are n codewords, the total probability for these types of collisions is $n \cdot 2^{-d}$, which is bounded by $\binom{n}{2} \cdot 2^{-d}$.

In the case of $i \neq j$, after generating t_i and s and fixing $c_{r_i,v}$ and $c_{r_j,u}$, exactly one value of t_j makes the equality hold. Therefore when generating t_j, the probability for a collision is $2^{-\kappa}$ for that specific pair. For fixed rows i and j there are at most $\binom{n}{2}$ different pairs $c_{r_i,v}, c_{r_j,u}$. Hence, the probability that a t_j makes one of these pairs collide, given fixed s and t_i, is at most $\binom{n}{2} \cdot 2^{-\kappa}$. Notice that the same probability bound for collisions holds when simply generating two rows of uniformly random values. Therefore, these types of collisions are roughly as likely in both scenarios and do not provide an advantage to the distinguisher. The overall probability of generating "bad" values that help the distinguisher is then still bounded by $\binom{n}{2} \cdot 2^{-d}$.

From this analysis, we can conclude that to achieve the same level of security κ_2 as in the 2-correlation robustness definition, the security parameter κ_n for n-correlation robustness should be chosen such that there exist codewords $c_1, \dots, c_n$ with minimum distance d, such that $\binom{n}{2} \cdot d \geq \kappa_2$. With a loss in a few bits of security for small n, we can simply take $d \geq \kappa_2$. In other words, the minimum distance of the codewords defines the equivalent security level to the 2-correlation robustness definition. Note that we require n codewords with length κ_n that have minimum distance d. To minimize κ_n, we can therefore use the Plotkin bound to find the smallest achievable value of κ_n for given d and n [Plo60]. These bounds are in fact achievable and we present concrete sets of codewords reaching the Plotkin bound in Appendix A.

Note that this reasoning agrees also with the original paper, where Walsh-Hadamard codes are used in the protocol [KK13]. The authors postulate that since Walsh-Hadamard codes have $d \leq \kappa_n/2$ then the security parameter κ_n should be roughly twice from that of the equivalent [IKNP03], the security of which can be based on 2-correlation robustness. Our insight, that we use to optimize communication in the Beaver triple generation protocol, is exactly that of using more efficient codes according to the Plotkin bound, when n is small, say 4 or 8.

We now finally present the protocol itself and give a proof of input privacy using our definition of n-correlation robustness.

Theorem 6. *The KK13 1-out-of-N correlated OT extension protocol in Protocol 3 is correct, assuming an n-correlation robust hash function H, secure PRG G and secure protocol for $\binom{2}{1}$-$\mathsf{OT}^{\kappa}_{\kappa}$.*

Proof sketch. We have that $\mathbf{q}_j = \mathbf{t}_{j,0} \oplus (\mathbf{s} \odot \mathbf{c}_{r_j})$. For $r_j = 1$, we therefore have $\mathbf{t}_{j,0} = \mathbf{q}_j \oplus (\mathbf{s} \odot \mathbf{c}_1)$ and the receiver computes the correct message x_j^1. For $r_j > 1$, we have that $H(\mathbf{q}_j \oplus (\mathbf{s} \odot \mathbf{c}_{r_j}) = H(\mathbf{t}_{j,0})$ and therefore, the receiver computes the correct message $x_j^{r_j}$. We also need to show that x_j^1 is pseudorandom. This follows

Protocol 3 Correlated 1-out-of-N OT extension protocol from [KK13]

Functionality: $\binom{n}{1}$-COT_ℓ^m

Setup: Security parameter κ where $n \leq \kappa$, n-correlation robust hash $H : \{0,1\}^\kappa \to \{0,1\}^\ell$, codewords $(\mathbf{c}_1, \ldots, \mathbf{c}_n)$ with minimum distance $d \leq \kappa$ and length κ, PRG $G : \{0,1\}^\kappa \to \{0,1\}^m$

Input: $\mathcal{S}$ has correlation offsets $c_j^i \in \{0,1\}^\ell$ for $j \in [m]$, $i \in \{2, \ldots, n\}$, $\mathcal{R}$ has m choice integers $\mathbf{r} = (r_1, \ldots, r_m)$ where $r_i \in [n]$

Result: $\mathcal{R}$ gets $(x_1^{r_1}, \ldots, x_m^{r_m})$, $\mathcal{S}$ gets m pseudorandom ℓ-bit values $x_1^1, \ldots, x_m^1$

```
    *Bootstrap phase*
 1: $\mathcal{S}$ generates random bit string $\mathbf{s} = (s_1, \ldots, s_\kappa)$
 2: $\mathcal{R}$ generates random $\kappa$-bit seed pairs $(k_i^0, k_i^1) \in \{0,1\}^{2\kappa}$ for $i \in [\kappa]$
 3: Perform $\binom{2}{1}$-$\mathsf{OT}_\kappa^\kappa$ with choices $s_i$ and messages $k_i^0, k_i^1$
    *Online phase*
 4: $\mathcal{R}$ generates a $m \times \kappa$ bit matrix $\mathbf{T}_0$ with columns $\mathbf{t}_0^i = G(k_i^0)$ and rows $\mathbf{t}_{j,0}$
 5: $\mathcal{R}$ computes a $m \times \kappa$ bit matrix $\mathbf{T}_1$ with rows $\mathbf{t}_{j,1} = \mathbf{t}_{j,0} \oplus \mathbf{c}_{r_j}$ and columns $\mathbf{t}_1^i$
 6: $\mathcal{R}$ sends $\mathbf{u}^i = \mathbf{t}_1^i \oplus G(k_i^1)$, $\mathcal{S}$ computes $\mathbf{q}^i = (s_i \cdot \mathbf{u}^i) \oplus G(k_i^{s_i})$
 7: $\mathcal{S}$ builds a $m \times \kappa$ bit matrix $\mathbf{Q}$ with columns $\mathbf{q}^i$ and rows $\mathbf{q}_j$   $\triangleright$ $\mathbf{q}^i = \mathbf{t}_{s_i}^i$ and
    $\mathbf{q}_j = \mathbf{t}_{j,0} \oplus (\mathbf{s} \odot (\mathbf{t}_{j,0} \oplus \mathbf{t}_{j,1})) = \mathbf{t}_{j,0} \oplus (\mathbf{s} \odot \mathbf{c}_{r_j})$
 8: $\mathcal{S}$ computes $x_j^1 = H(\mathbf{q}_j \oplus (\mathbf{s} \odot \mathbf{c}_1))$, $x_j^i = x_j^1 + c_j^i$ for $2 \leq i \leq n$, $j \in [m]$
 9: $\mathcal{S}$ sends $y_j^r = x_j^r \oplus H(\mathbf{q}_j \oplus (\mathbf{s} \odot \mathbf{c}_r))$ to $\mathcal{R}$ for $2 \leq r \leq n$, $j \in [m]$
10: Receiver computes $x_j^{r_j} = y_j^{r_j} \oplus H(\mathbf{t}_{j,0})$   $\triangleright$ $y_j^1 = 0$ and is not sent
11: **return** $(x_j^1, x_j^{r_j})$ for $j \in [m]$
```

from a similar argument as in Lemma 2, modified to the generalized correlation robustness definition. □

Theorem 7. *The correlated 1-out-of-N OT extension protocol presented in Protocol 3 is computationally input-private, assuming a secure PRG G, an n-correlation-robust hash function H and a computationally secure protocol for $\binom{2}{1}$-$\mathsf{OT}_\kappa^\kappa$.*

Proof. The proof is very similar in structure to the input privacy proof of ALSZ13 correlated OT protocol in Theorem 3.

Corrupted sender. Simulation for the bootstrapping phase is identical to the proof of ALSZ13 correlated OT protocol in Theorem 3. We additionally need to simulate the message $\mathbf{u}^i$ to the corrupted sender. This can be perfectly simulated by generating a uniformly random element from $\{0,1\}^m$, since

$$\mathbf{u}^i = G(k_i^0) \oplus G(k_i^1) \oplus \mathbf{c}^i$$

where $\mathbf{c}^i$ consists of the bits $(\mathbf{c}_{r_1}[i], \ldots, \mathbf{c}_{r_m}[i])$. As $\mathbf{u}^i$ is masked with $G(k_i^{s_i - 1})$, it is computationally indistinguishable from uniform randomness for the adversary.

Corrupted receiver. The simulation of the bootstrapping phase follows the proof of Theorem 3. Now, we need to simulate the messages y_j^r for $2 \leq r \leq n$. First, we consider the case $r = r_j$. Then we have that

$$\begin{aligned} y_j^{r_j} &= x_j^{r_j} \oplus H(\mathbf{q}_j \oplus (\mathbf{s} \odot \mathbf{c}_{r_j})) = x_j^{r_j} \oplus H(\mathbf{t}_{j,0} \oplus (\mathbf{s} \odot (\mathbf{c}_{r_j} \oplus \mathbf{c}_{r_j}))) \\ &= x_j^{r_j} \oplus H(\mathbf{t}_{j,0}) = (x_j^1 + H(\mathbf{t}_{j,0} \oplus (\mathbf{s} \odot (\mathbf{c}_1 \oplus \mathbf{c}_{r_j})))) \oplus H(\mathbf{t}_{j,0}) \ . \end{aligned}$$

Note, that $r_j \neq 1$. Here, we can therefore use the n-correlation robustness property of H, to conclude that the values $H(\mathbf{t}_{j,0} \oplus (\mathbf{s} \odot (\mathbf{c}_1 \oplus \mathbf{c}_{r_j})))$ are computationally indistinguishable from uniform randomness for all $j \in [m]$, since $\mathbf{t}_{j,0}$ are pseudorandom. A simple hybrid argument suffices to replace the pseudorandom $\mathbf{t}_{j,0}$ with a uniformly random one (due to security of the PRG). Therefore, for the case $r_j = r$, we can simulate y_j^r perfectly by generating a uniformly random element.

Now consider the case $r \neq r_j$. We now have

$$y_j^r = x_j^r \oplus H(\mathbf{q}_j \oplus (\mathbf{s} \odot \mathbf{c}_r)) = x_j^{r_j} \oplus H(\mathbf{t}_{j,0} \oplus (\mathbf{s} \odot (\mathbf{c}_r \oplus \mathbf{c}_{r_j}))) \ .$$

Similarly as before, we use the n-correlation robustness assumption to conclude that y_j^r can be simulated perfectly by generating a uniformly random value. This concludes the proof, as we have simulated all messages perfectly for a corrupted receiver. □

3.4 Implementing the base OT

We briefly comment in this section about our assumption of a secure $\binom{2}{1}$-$\mathsf{OT}_\kappa^\kappa$ protocol, that we have made in the OT extension protocols' security proofs. As we have made efforts to avoid the random oracle model, a suitable instantiation would be for example the protocol from [NP01], which relies on the standard decisional Diffie-Hellman assumption. However, the protocol is not fully simulatable and therefore, we would lose security guarantees in the concurrent model. Alternatively, the efficient protocol of [CO15] is UC-secure against malicious adversaries, but relies on a programmable random oracle. It is possible that a passively-secure adaptation of the protocol could be shown secure under weaker assumptions. The [ALSZ13] paper also presents a fully simulatable OT protocol, that relies on the pseudorandom output of key derivation functions. Finally, the actively UC-secure protocol of [Lin08] is fully simulatable and relies only on standard computational assumptions, and is therefore the most conservative choice in terms of security, but also the most costly in terms of performance.

As we actually only require a random OT functionality in the bootstrapping phase (the values k_i^0, k_i^1 are generated randomly), then we might be able to use more efficient protocols by relying on an input privacy argument similarly to how we have approached the OT extension protocols in this section. However, this requires more careful analysis also in the current proofs. Overall, we leave the best choice of a base OT protocol regarding both efficiency and the required security assumptions as future work. The overall efficiency of the precomputation phase is not much affected by this choice, since the cost of the base OT-s amortizes away as we can extend them to a much large amount of oblivious transfers.

4 Beaver triple generation

This section is dedicated to Beaver triples, which form the basis for all protocols in our implemented two-party protection domain. We first describe what Beaver triples are and describe efficient protocols based on oblivious transfer extension for generating them, which use the protocols we have described in the previous section. The rest of the section focuses on efficient practical implementations of these methods. Especially, we provide some novel communication optimizations based on the $\binom{n}{1}$-OT extension protocol of [KK13]. We describe our implementation of the triple generation protocols and describe the practical challenges we discovered. Finally, we present benchmark results for our implementation.

4.1 Beaver triples

The famous paper by D. Beaver in 1991 proposed a method to reduce the round-complexity of BGW-style protocols [Bea91]. The general idea is to precompute the multiplication gates for an arithmetic circuit using random inputs (and receiving random results). In the online phase, these precomputed random multiplication triples can be used to do the actual multiplication more efficiently.

A Beaver triple is a multiplication triple $[\![a]\!] \cdot [\![b]\!] = [\![c]\!]$, where the values of a, b are random and $c = a \cdot b$. For our purposes, we consider the additive secret sharing scheme for the Beaver triple values, although any linear secret sharing scheme could be used. In the three-party setting with passive security, performing a multiplication on additively shared integers can be done using a simple one-round input-private protocol [BNTW12]. In the two-party setting, Beaver triples enable a similar input-private protocol with only one round, assuming that the parties already share a random Beaver triple. The multiplication protocol itself is presented in Section 5, where we describe our full two-party protocol suite (Protocol 14).

The multiplication protocol makes use of a single Beaver triple to perform a multiplication of two secret-shared inputs. However, the triples can be computed by the parties in an offline precomputation phase, since they are independent of the inputs. Therefore, our two-party protocol suite adopts the *offline-online* model of computation [DN07]. The actual computations on secret data are performed in an online phase, which we outline in Section 5. However, all protocols in the online phase require Beaver triples, and as such, we need to precompute them beforehand in the offline phase. The offline phase itself is more costly in terms of computation and communication, which is why the main focus of this work is in optimizing the generation of Beaver triples.

We define now what security properties we require from the generated Beaver triples. Intuitively, the triple values should be random and a party should have no information about the other party's shares of the triple. We therefore define an ideal functionality for a protocol run between parties $\mathcal{CP}_1$, $\mathcal{CP}_2$ which takes no inputs and outputs a triple $[\![a]\!] \cdot [\![b]\!] = [\![c]\!]$ shared between the parties.

Definition 12. *An ideal functionality $\mathcal{F}_B$ for computing Beaver triples is defined as follows. $\mathcal{F}_B$ takes no inputs and generates $a \leftarrow \mathbb{Z}_{2^\ell}$, $b \leftarrow \mathbb{Z}_{2^\ell}$ and computes $c = a \cdot b$. Then $\mathcal{F}_B$ computes $([\![a]\!]_1, [\![a]\!]_2) \leftarrow \mathsf{Share}(a)$, $([\![b]\!]_1, [\![b]\!]_2) \leftarrow \mathsf{Share}(b)$ and $([\![c]\!]_1, [\![c]\!]_2) \leftarrow \mathsf{Share}(c)$, where $\mathsf{Share}(\cdot)$ belongs to a perfectly secure $(2,2)$-secret sharing scheme according to Def. 2. Finally, $\mathcal{F}_B$ outputs $([\![a]\!], [\![b]\!], [\![c]\!])$ secret-shared between parties $\mathcal{CP}_1$ and $\mathcal{CP}_2$.*

Definition 13. *We say π is a secure Beaver triple generation protocol, if it is computationally secure w.r.t to the ideal functionality $\mathcal{F}_B$ defined in Def. 12.*

Analogously to additive multiplication, we can perform bitwise conjunction of bitwise shared integers, using a triple of the form $[\![a]\!] \wedge [\![b]\!] = [\![c]\!]$ in bitwise sharing. The conjunction protocol and required security properties for the triple are identical to the additive multiplication case. Notice also that the ideal functionality defined in Def. 12 also holds for conjunction triples, if we consider triples over $\mathbb{Z}_2$. A conjunction triple for $\mathbb{Z}_2^\ell$ is then trivial to construct by concatenating the shares of single bit triples together locally.

Next, we discuss how to implement a secure protocol for Beaver triple generation using oblivious transfer extension.

4.2 Computing multiplication triples with oblivious transfer

This section leads to presenting a secure protocol for generating Beaver triples as Protocol 6, which was presented in [DSZ15].

4.2.1 Gilboa's protocol

We first require a sub-protocol for computing the product of two integers $x, y \in \mathbb{Z}_{2^\ell}$, where x is known by $\mathcal{CP}_1$ and y by $\mathcal{CP}_2$. The protocol in Protocol 4 was introduced by N. Gilboa [Gil99]. In the original paper, the protocol is used as a building block for two parties to generate a shared RSA key that allows threshold decryption. The original protocol uses $\binom{2}{1}$-OT to accomplish this, however it can be naturally adjusted to use $\binom{2}{1}$-COT instead as presented in [DSZ15]. We present the more efficient $\binom{2}{1}$-COT version as Protocol 4.

Protocol 4 Gilboa's protocol for multiplying ℓ-bit integers held by different parties using $\binom{2}{1}$-COT_ℓ

```
Functionality: $\mathcal{F}(x, y) = [\![z]\!]$, such that $z = xy$
Input: $\mathcal{CP}_1$ inputs $x$, $\mathcal{CP}_2$ inputs $y$
Result: Additively shared result $[\![z]\!]$
1: for $i \in [\ell]$ do                    ▷ Perform a $\binom{2}{1}$-$\mathsf{COT}$ for each bit
2:     $\mathcal{CP}_1$ fixes a correlation offset $c_i = x \cdot 2^{i-1}$
3:     Perform $\binom{2}{1}$-$\mathsf{COT}_\ell$ with correlation offset $c_i$ from $\mathcal{CP}_1$ and choice bit $y[i]$
   from $\mathcal{CP}_2$. $\mathcal{CP}_1$ receives $s_{i,0}$ and $\mathcal{CP}_2$ receives $t_i = s_{i,0} + y[i] \cdot c_i$.
4: end for
5: $\mathcal{CP}_1$ fixes $[\![z]\!]_1 = -\sum_{i=1}^{\ell} s_{i,0}$
6: $\mathcal{CP}_2$ fixes $[\![z]\!]_2 = \sum_{i=1}^{\ell} t_i = \sum_{i=1}^{\ell} y[i]\,(x \cdot 2^{i-1}) + s_{i,0}$
7: return $[\![z]\!]$
```

We stress that all computations performed in the protocol are modulo 2^ℓ. We now show correctness and input privacy for Gilboa's protocol.

Theorem 8. *Gilboa's protocol for multiplying ℓ-bit integers held by different parties using $\binom{2}{1}$-COT_ℓ in Protocol 4 is correct, assuming a correct protocol for $\binom{2}{1}$-COT_ℓ^ℓ.*

Proof. The correctness of the protocol is straightforward as we can consider the binary representation of y as $y = \sum_{i=1}^{\ell} y[i] \cdot 2^{i-1}$. Then we have

$$\begin{aligned}
[\![z]\!]_2 + [\![z]\!]_1 &= \sum_{i=1}^{\ell} t_i - \sum_{i=1}^{\ell} s_{i,0} \\
&= \sum_{i=1}^{\ell} y[i]\left(x \cdot 2^{i-1}\right) + s_{i,0} - \sum_{i=1}^{\ell} s_{i,0} \\
&= x \sum_{i=1}^{\ell} y[i] \cdot 2^{i-1} \\
&= xy \ .
\end{aligned}$$

We also have correctness of the output distribution, since the correctness of $\binom{2}{1}$-COT_ℓ^ℓ assures pseudorandom distribution for each $s_{i,0}$. □

Theorem 9. *Protocol 4 is computationally input-private assuming a correct and computationally input-private protocol for $\binom{2}{1}$-COT_ℓ^ℓ.*

Proof. Since the only communication is performed in the $\binom{2}{1}$-COT execution, input privacy follows directly from the input privacy of $\binom{2}{1}$-COT, since we have an ordered composition. □

We note an optimization trick that can be used to reduce communication in Protocol 4, which was pointed out in [DSZ15]. Notice that when performing $\binom{2}{1}$-COT_ℓ for calculation of the i-th bit of the result, the correlation offset c_i only retains the $\ell - i + 1$ least significant bits of x, and the rest are zero, since they are discarded modulo 2^ℓ. Therefore, we can reformulate the protocol to use $c_i = x \bmod 2^{\ell-i+1}$ and perform a $\binom{2}{1}$-$\mathsf{COT}_{\ell-i+1}$ instead. Later, the t_i and s_i values need to be adjusted accordingly to produce the correct result. This optimization does not affect security if we use the ALSZ13 $\binom{2}{1}$-COT protocol (Protocol 1), since we have shown that performing $\binom{2}{1}$-COT_t for $t < \ell$ is straightforward under the same security assumptions.

4.2.2 Secure resharing

For Beaver triple generation, we require a secure resharing step to ensure that the outputs of the protocol are uniformly random and that we can perfectly simulate these shares. The Reshare protocol in Protocol 5 is also an invaluable tool for transforming input-private protocols into secure ones as we discuss in Section 5.

Protocol 5 Reshare of ℓ-bit additive types

Functionality: $\mathcal{F}(\llbracket x \rrbracket) = \llbracket x' \rrbracket$, where $x = x'$
Input: Shared value $\llbracket x \rrbracket$
Result: Shared value $\llbracket x' \rrbracket$

```
1: Both $\mathcal{CP}_i$ generate $r_i \leftarrow \mathbb{Z}_{2^\ell}$ and send $r_i$ to $\mathcal{CP}_j$
2: Both $\mathcal{CP}_i$ compute $\llbracket x' \rrbracket_i = \llbracket x \rrbracket_i - r_i + r_j$
3: return $\llbracket x' \rrbracket$
```

Theorem 10. *The* Reshare *protocol in Protocol 5 is perfectly secure.*

Proof. We can perfectly simulate the message r_j for corrupt $\mathcal{CP}_i$ by generating $\overline{r_i} \leftarrow \mathbb{Z}_{2^\ell}$ and computing $\overline{r_j} = \llbracket x' \rrbracket_i - \llbracket x \rrbracket_i + \overline{r_i}$ from ideal functionality's output of $\llbracket x' \rrbracket_i$. Thus, the values $\overline{r_i}$, $\overline{r_j}$ provide a consistent view to the environment, since

$$\llbracket x' \rrbracket_j = \llbracket x \rrbracket_j - \overline{r_j} + \overline{r_i} = x - \llbracket x \rrbracket_i - \llbracket x' \rrbracket_i + \llbracket x \rrbracket_i - \overline{r_i} + \overline{r_i} = x - \llbracket x' \rrbracket_i \ .$$

Also, since r_i is randomly generated, the output share $\llbracket x' \rrbracket_i$ is uniformly random and independent from $\llbracket x \rrbracket_i$ due to Lemma 1 in Section 2.3. □

Note that an analogous protocol can be constructed for bitwise shared values, by computing $\llbracket x' \rrbracket_i = \llbracket x \rrbracket_i \oplus r_i \oplus r_j$ as the new share for $\mathcal{CP}_i$. The security proof for the bitwise case also follows the same reasoning.

4.2.3 Secure Beaver triple generation

We now turn to our Beaver triple generation protocol. The main idea is simple. First, the parties can randomize their shares of $[\![a]\!]$ and $[\![b]\!]$ locally, which exactly matches what is done by the ideal functionality $\mathcal{F}_B$ in Def. 12[10]. It remains to calculate shares for $[\![c]\!]$, such that $c = ab$. We have that

$$([\![a]\!]_1 + [\![a]\!]_2) \cdot ([\![b]\!]_1 + [\![b]\!]_2) = [\![a]\!]_1 \cdot [\![b]\!]_1 + [\![a]\!]_1 \cdot [\![b]\!]_2 + [\![a]\!]_2 \cdot [\![b]\!]_1 + [\![a]\!]_2 \cdot [\![b]\!]_2 \ .$$

The multiplications $[\![a]\!]_i \cdot [\![b]\!]_i$ can be computed locally by $\mathcal{CP}_i$. For the multiplications $[\![a]\!]_1 \cdot [\![b]\!]_2$ and $[\![a]\!]_2 \cdot [\![b]\!]_1$, we can directly use Gilboa's protocol presented before as Protocol 4. Additionally, for full security we have to explicitly reshare the output shares. We present the Beaver triple generation protocol as Protocol 6.

Protocol 6 Computation of ℓ-bit multiplication triples

Functionality: $\mathcal{F}_B$ according to Def. 12 for ℓ-bit values
Input: No input
Result: Beaver triple $[\![a]\!] \cdot [\![b]\!] = [\![c]\!]$

```
1: $\mathcal{CP}_i$ generate uniformly random values $[\![a']\!]_i \leftarrow \mathbb{Z}_{2^\ell}$, $[\![b']\!]_i \leftarrow \mathbb{Z}_{2^\ell}$
2: The parties compute $[\![u]\!] = [\![a']\!]_1 \cdot [\![b']\!]_2$ using Protocol 4
3: The parties compute $[\![v]\!] = [\![a']\!]_2 \cdot [\![b']\!]_1$ using Protocol 4
4: $\mathcal{CP}_i$ fixes $[\![c']\!]_i = [\![a']\!]_i \cdot [\![b']\!]_i + [\![u]\!]_i + [\![v]\!]_i$
5: $[\![a]\!] \leftarrow \mathsf{Reshare}([\![a']\!])$, $[\![b]\!] \leftarrow \mathsf{Reshare}([\![b']\!])$ and $[\![c]\!] \leftarrow \mathsf{Reshare}([\![c']\!])$
6: return $[\![a]\!], [\![b]\!], [\![c]\!]$
```

Theorem 11. *The Beaver multiplication triple generation algorithm in Protocol 6 is correct w.r.t $\mathcal{F}_B$ as defined in Def. 12.*

Proof. For correctness, we require that $[\![c']\!] = [\![a']\!] \cdot [\![b']\!]$, since then also $[\![c]\!] = [\![a]\!] \cdot [\![b]\!]$. From the correctness of Protocol 4, we have that $[\![u]\!]_1 + [\![u]\!]_2 = [\![a']\!]_1 \cdot [\![b']\!]_2$ and $[\![v]\!]_1 + [\![v]\!]_2 = [\![a']\!]_2 \cdot [\![b']\!]_1$. Then

$$\begin{aligned}
[\![c']\!] &= [\![c']\!]_1 + [\![c']\!]_2 \\
&= [\![a']\!]_1 \cdot [\![b']\!]_1 + [\![u]\!]_1 + [\![v]\!]_1 + [\![a']\!]_2 \cdot [\![b']\!]_2 + [\![u]\!]_2 + [\![v]\!]_2 \\
&= [\![a']\!]_1 \cdot [\![b']\!]_1 + [\![a']\!]_1 \cdot [\![b']\!]_2 + [\![a']\!]_2 \cdot [\![b']\!]_1 + [\![a']\!]_2 \cdot [\![b']\!]_2 \\
&= ([\![a']\!]_1 + [\![a']\!]_2) \cdot ([\![b']\!]_1 + [\![b']\!]_2) \\
&= [\![a']\!] \cdot [\![b']\!] \ .
\end{aligned}$$

[10] $\mathcal{F}_B$ first generates $a \leftarrow \mathbb{Z}_{2^\ell}$, then does $([\![a]\!]_i, [\![a]\!]_2) \leftarrow \mathsf{Share}[\![a]\!]$. For additive sharing, this is equivalent to generating random shares $[\![a]\!]_1 \leftarrow \mathbb{Z}_{2^\ell}$, $[\![a]\!]_2 \leftarrow \mathbb{Z}_{2^\ell}$, since then $[\![a]\!]_1 + [\![a]\!]_2$ is also a uniformly random value.

Also, we have that the outputs are of the correct distribution, since shares of $[\![a]\!]$ and $[\![b]\!]$ are generated uniformly randomly. Shares of $[\![c]\!]$ are also computationally indistinguishable from uniform randomness, due to the correctness of Protocol 4, which ensures $[\![u]\!]_i$ and $[\![v]\!]_i$ are pseudorandom, independently of $[\![a]\!]_i$ and $[\![b]\!]_i$. □

Theorem 12. *The Beaver multiplication triple generation algorithm in Protocol 6 is computationally secure w.r.t $\mathcal{F}_B$ defined in Def. 12.*

Proof sketch. The security follows from the composition of an input-private and secure protocol from [BLLP14]. The correctness of the correlated OT used in Gilboa's protocol suffices to show the output predictability of the whole composition as required in [BLLP14]. In short, since Protocol 4 is input-private, the shares of $[\![a']\!]_i$ and $[\![b']\!]_i$ are not leaked to the other party. Adding a reshare step ensures that the output shares are uniformly random and independent as they are generated also in $\mathcal{F}_B$. □

Note that both multiplications $[\![a]\!]_1 \cdot [\![b]\!]_2$ and $[\![a]\!]_2 \cdot [\![b]\!]_1$ can be performed independently in parallel, since they have no data dependencies. This does not affect security, since, in our model, the composition of two input-private protocols without data dependencies is trivially input-private. Also, to even the computational load of the computing parties, the parties can switch their roles in Protocol 4 for the second multiplication. This means the sender and receiver roles for oblivious transfer are reversed, which optimizes computation load, since the sender does more work in most oblivious transfer protocols, including the oblivious transfer extension protocols we have discussed.

4.3 Computing bitwise conjunction triples with random-OT

We present the protocol from [DSZ15] for generating conjunction triples using $\binom{2}{1}$-ROT. Note that conjunction triples are actually 1-bit multiplication triples, so we can also generate these using Protocol 6. However, bit-triples can be generated using the more efficient random OT instead of correlated OT. First, we present a sub-routine with no inputs for generating bit values $ab = u \oplus v$, where (a, u) are received by $\mathcal{CP}_1$ and (b, v) by $\mathcal{CP}_2$. We require from the ideal functionality that a and b are generated uniformly randomly, and then ab is securely secret-shared, that is, u and v are individually uniformly random.

Protocol 7 Computing $(a, u), (b, v)$, where $ab = u \oplus v$

Functionality: $\mathcal{F}(\perp, \perp) = ((a, u), (b, v))$ $ab = u \oplus v$ for 1-bit values
Input: No input
Result: $\mathcal{CP}_1$ gets (a, u), $\mathcal{CP}_2$ gets (b, v)

```
1: $\mathcal{CP}_1$ generates $a \leftarrow \{0, 1\}$
2: Perform $\binom{2}{1}$-$\mathsf{ROT}_1$ with choice bit $a$ from $\mathcal{CP}_1$. $\mathcal{CP}_1$ gets $x_a$ and $\mathcal{CP}_2$ gets
   pseudorandom $x_0$, $x_1$.
3: **return** $((a, x_a), (x_0 \oplus x_1, x_0))$
```

Theorem 13. *The protocol for generating random bit values $ab = u \oplus v$ between two parties in Protocol 7 is correct assuming a correct protocol for $\binom{2}{1}$-ROT_1 that does not leak the other message to the receiver, as in Theorem 5.*

Proof. From the calls to Protocol 7, we have that $a_1b_2 = u_1 \oplus v_2$ and $a_2b_1 = u_2 \oplus v_1$. Correctness follows from:

$$\begin{aligned}(a_1 \oplus a_2) \wedge (b_1 \oplus b_2) &= a_1b_1 \oplus a_1b_2 \oplus a_2b_1 \oplus a_2b_2 \\ &= c_1 \oplus u_1 \oplus v_1 \oplus a_1b_2 \oplus a_2b_1 \oplus c_2 \oplus u_2 \oplus v_2 \\ &= c_1 \oplus c_2 \ .\end{aligned}$$

For the output distributions, we first have that a is generated uniformly randomly. From the correctness of $\binom{2}{1}$-ROT_1, we have that $v = x_0$ is pseudorandom, and input privacy of $\binom{2}{1}$-ROT_1 ensures x_0 is independent of a. Correctness of $\binom{2}{1}$-ROT_1 also ensures that $b = x_0 \oplus x_1$ is pseudorandom and independent of $v = x_0$, as x_0 and x_1 are independently pseudorandom. Also, b is independent of a due to input privacy of $\binom{2}{1}$-ROT_1.

Finally, $u = x_a$ is pseudorandom from correctness of $\binom{2}{1}$-ROT_1, it is independent of a due to input privacy of $\binom{2}{1}$-ROT_1 and it is independent of $b = x_0 \oplus x_1$ due to the guarantee that x_{1-a} is not leaked to the receiver in $\binom{2}{1}$-ROT_1.

□

The protocol is also trivially input-private as there are no inputs. Using two invocations of this sub-routine, we can construct a bit conjunction triple generation protocol, presented as Protocol 8.

Protocol 8 Computation of bit conjunction triples

Functionality: $\mathcal{F}_B$ according to Def. 12 for 1-bit values
Input: No input
Result: Conjunction triple $[\![a]\!] \wedge [\![b]\!] = [\![c]\!]$

```
1: Parties run Protocol 7. $\mathcal{CP}_1$ gets $(a_1, u_1)$ and $\mathcal{CP}_2$ gets $(b_2, v_2)$
2: Parties run Protocol 7 with reversed roles. $\mathcal{CP}_2$ gets $(a_2, u_2)$ and $\mathcal{CP}_1$ gets $(b_1, v_1)$
3: Both $\mathcal{CP}_i$ fix $[\![a']\!]_i = a_i$, $[\![b']\!]_i = b_i$, $[\![c']\!]_i = a_i b_i \oplus u_i \oplus v_i$
4: $[\![a]\!] \leftarrow \mathsf{Reshare}([\![a']\!])$, $[\![b]\!] \leftarrow \mathsf{Reshare}([\![b']\!])$ and $[\![c]\!] \leftarrow \mathsf{Reshare}([\![c']\!])$
5: return $[\![a]\!], [\![b]\!], [\![c]\!]$
```

Theorem 14. *The bit conjunction triple generation algorithm in Protocol 8 is correct w.r.t $\mathcal{F}_B$ defined in Def. 12.*

Proof. From the calls to Protocol 7, we have that $a_1 b_2 = u_1 \oplus v_2$ and $a_2 b_1 = u_2 \oplus v_1$. Correctness follows from:

$$\begin{aligned}(a_1 \oplus a_2) \wedge (b_1 \oplus b_2) &= a_1 b_1 \oplus a_1 b_2 \oplus a_2 b_1 \oplus a_2 b_2 \\ &= c_1 \oplus u_1 \oplus v_1 \oplus a_1 b_2 \oplus a_2 b_1 \oplus c_2 \oplus u_2 \oplus v_2 \\ &= c_1 \oplus c_2 \ .\end{aligned}$$

Concerning the distribution of outputs, we have that shares of $[\![a']\!]$ and $[\![b']\!]$ are pseudorandom due to correctness of Protocol 7. Also, since u_i and v_i are pseudorandom and produced by separate invocations of Protocol 7, also $u_i \oplus v_i$ is pseudorandom. Therefore, shares of $[\![c']\!]$ are also pseudorandom, since u_i is independent of a_i, v_i is independent of b_i and also a_i, b_i are independent as they are generated from separate invocations of correct Protocol 7. □

Theorem 15. *The bit conjunction triple generation algorithm in Protocol 8 is computationally secure w.r.t $\mathcal{F}_B$ defined in Def. 12.*

Proof sketch. Security follows similarly as in the case of the multiplication triple protocol Theorem 5.3.2 from composition, due to the input privacy and correctness of Protocol 7. □

4.4 Optimizing communication in Beaver triple generation

We now discuss how to further optimize the communication of the Beaver triple generation protocol presented as Protocol 6. The rest of this section focuses specifically on multiplication triples, since the protocol for generating conjunction triples is already more efficient, by using random OT. As such, the optimizations discussed here are specific to methods based on correlated OT. Our main insight is in fact using the KK13 protocol and $\binom{n}{1}$-OT for implementing Protocol 6, instead of $\binom{2}{1}$-OT[11].

We first establish a baseline protocol for triple generation and introduce our optimizations with respect to this baseline. Throughout this comparison, we assume a strong security parameter $\kappa = 128$ for long-term security.

4.4.1 Baseline

We consider the baseline protocol to be the triple generation protocol presented in Protocol 6 using the ALSZ13 $\binom{2}{1}$-COT extension. We assume the two multiplications that require Gilboa's protocol are done with reversed roles to even the computational load of the parties. That is, one invocation of Protocol 4 has $\mathcal{CP}_1$ as the OT sender and $\mathcal{CP}_2$ as the receiver, and vice versa for the second invocation. Notice also that these computations can be be performed independently in parallel. We immediately note that our implementation currently performs the two invocations sequentially, however, the benchmarks clearly show this as a valuable optimization for the future. We discuss this topic further in the Section 4.6.

We also include the optimization outlined in Section 4.2.1. This approach theoretically saves a total of $\frac{\ell\cdot(\ell-1)}{2}$ bits of communication in Gilboa's protocol (Protocol 4) for ℓ-bit integers if we use ALSZ13 $\binom{2}{1}$-COT_t, since the y_j value has length t in Protocol 1. To the best of our knowledge, this is exactly the protocol used for multiplication triple generation in the ABY framework [DSZ15], disregarding the concrete instantiations used for the hash function and PRG. As such, we can also later make a direct comparison with the benchmark results from [DSZ15].

4.4.2 Using 1-out-of-N OT for Beaver triple generation

1-out-of-2^{L_i} Using the KK13 OT extension protocol (Protocol 3), we can perform $\binom{N}{1}$-COT for $N > 2$. We notice that this allows us to calculate more than a single bit in the multiplication for Gilboa's protocol using a single OT. Namely, we can do an $\binom{2^L}{1}$-OT to calculate L bits of the multiplication to get better communication complexity than in the baseline protocol. In addition, we can vary the value of L in successive OT-s for more communication cut-offs. We present this approach as a more generalized version of Gilboa's protocol in Protocol 9.

[11]Studying the possibilities for optimizing communication by using the KK13 protocol has been the joint work of the thesis author, supervisors and Claudio Orlandi from Aarhus University.

Protocol 9 Gilboa's protocol for multiplying ℓ-bit integers held by different parties using $\binom{2^{L_i}}{1}$-COT

Functionality: $\mathcal{F}(x, y) = [\![z]\!]$, such that $z = xy$
Setup: Values $L_1, \ldots, L_k$, such that $L_i \geq 1$ and $\sum_{i=1}^{k} L_i = \ell$
Input: $\mathcal{CP}_1$ inputs x, $\mathcal{CP}_2$ inputs y
Result: Additively shared result $[\![z]\!]$

```
1: $t = 0$
2: **for** $i \in [k]$ **do**                    ▷ Perform $\binom{2^{L_i}}{1}$-$\mathsf{COT}$ for $L_i$-bit chunks
3:     $\mathcal{CP}_1$ fixes correlation offsets $c_i^j = x \cdot 2^{t+j-2}$ for $j \in 2, \ldots, L_i$
4:     $\mathcal{CP}_2$ fixes choice index $a_i = \left(\sum_{j=t+1}^{t+L_i} 2^{j-t-1} y[j]\right) + 1$
5:     Perform $\binom{2^{L_i}}{1}$-$\mathsf{COT}_\ell$ with correlation offsets $c_i^j$ from $\mathcal{CP}_1$ and choice index
       $a_i$ from $\mathcal{CP}_2$. $\mathcal{CP}_1$ receives $s_{i,1}$ and $\mathcal{CP}_2$ receives $t_i$
6:     $t = t + L_i$
7: **end for**
8: $\mathcal{CP}_1$ fixes $[\![z]\!]_1 = -\sum_{i=1}^{k} s_{i,1}$
9: $\mathcal{CP}_2$ fixes $[\![z]\!]_2 = \sum_{i=1}^{k} t_i$
10: **return** $[\![z]\!]$
```

The input privacy and correctness of output distributions follows as in the proof of original Gilboa's protocol, but assuming an input-private and correct protocol for each $\binom{L_i}{1}$-COT $i \in [k]$. We now give intuition that the computed result is in fact correct by going through a simple example of a two-bit multiplication with one iteration $L_1 = 2$. The idea is, that the receiver now chooses the correct multiplication value based on 2 bits of his share. For two bits, we have that the receiver's choice is $a \in [4]$, the binary representation of which matches the two bits exactly. Notice, that the required result of the protocol is exactly $(a - 1) \cdot x$. Now, the sender fixes the correlation offsets respectively, so that the receiver would get the correct result, either $0 \cdot x$, x, $2x$ or $3x$, masked with the pseudorandom $s_{i,1}$. For $a = 1$, the receiver gets exactly $s_i, 1$ and the shares thus cancel out in the result, giving 0. For larger bit-length values, we can continue this process iteratively, with the receiver obliviously choosing values $(a - 1) \cdot x \cdot 2^t$, where a represents the current bits after position t.

The reduction in communication complexity, when compared to the baseline protocol, comes from the fact that the sender only sends one message (matrix row) of length κ_1 in the $\binom{2^{L_i}}{1}$-COT protocol for computing L_i bits of the result. In the baseline, we send a κ_2-bit message for each bit in the multiplication, however, we also have that $\kappa_2 < \kappa_1$. Additionally, the sender has to send a larger amount of ℓ-bit messages in the $\binom{n}{1}$-COT protocols, namely $n - 1$. However, since $\ell < \kappa_2$ for computing common bit-length multiplication triples, we can find a suitable choice of parameters for actually improving on communication complexity compared to the baseline.

The same optimization for reducing the lengths of the OT messages applies in the $\binom{n}{1}$-OT case also, as we can consider only the least significant bits of x that are kept in the correlation offset. For using the KK13 protocols, we use codewords that achieve the Plotkin bound, as we discussed in Section 3.3. We have constructed the required sets of codewords by hand and they are presented explicitly in Appendix A. We summarize the code parameters in Table 1.

Table 1: Parameters of codes used in our KK13 implementation.

Minimum distance d	**Number of codewords** n	**Code length (bits)** (κ)
128	2	128
128	4	192
128	8	224
128	16	240

The code lengths correspond to the required security parameter κ of the KK13 protocol and for performing $\binom{n}{1}$-OT, we need n distinct codewords with pair-wise minimum distance $d = 128$. We did not need to use a larger number of messages than 16 with our chosen parameters, but it is possible to perform up to $\binom{256}{1}$-OT using for example Walsh-Hadamard codes with length 256 [KK13].

To get the best communication gain from the 1-out-of-2^{L_i} approach, we can compute optimal values L_i in terms of total communication and do the multiplication with $\binom{2^{L_i}}{1}$-COT as in Protocol 9. We have computed optimal L_i values naively, by simply computing the total communication cost for all combinations and picking the best solution. We present optimal values for different bit-length triples below in Table 2. We perform our analysis and benchmarks for these common bit-length values, since we require triples of these bit-lengths for the online phase protocols in Section 5.

Note that we use the security parameter required by the largest L_i value for all the OT-s and we computed optimal L_i values assuming this. Theoretically, we could use different security parameters for the different invocations, but that would make the implementation quite a bit more complicated for performing these OT-s in a single batch, as the matrix rows generated by the PRG-s would have different lengths and would be inconvenient to transpose.

Also, note that we assume here, that the message length ℓ in $\binom{L_i}{1}$-COT_ℓ is always a multiple of 8, in other words a full byte, in accordance with our current implementation. Again, this is an implementation difficulty, in having to pack

Table 2: Optimal L_i values in terms of total communication when using 1-out-of-2^{L_i} strategy using Protocol 9.

Triple length ℓ	**L_i values**	**Required KK13 security parameter κ**	**Communication (bits)**
8	(4,4)	240	1440
16	(4,4,4,4)	240	3360
32	(2,3,3,3,3,3,3,3,3,3,3)	224	7948
64	(2, ..., 2)	192	19200

different bit-length messages together into a byte array, and we currently have not implemented this. The communication costs also reflect this redundancy. With bit-level packing, better communication can be achieved in all cases. However, our computations show that with bit-level packing, the relative reduction in communication from using the KK13 methods is even slightly larger, although the baseline protocol's communication is also reduced.

1-out-of-4^{L_i} Another observation is that we can calculate one bit in the sum $[\![a]\!]_1 \cdot [\![b]\!]_2 + [\![a]\!]_2 \cdot [\![b]\!]_1$ using a single call to $\binom{4}{1}$-OT, instead of 2 calls to $\binom{2}{1}$-OT. Similarly to the previous, we can do 1-out-of-4^{L_i} to calculate also L_i bits of the result at once. This means we do yet another modified version of the Gilboa's protocol by choosing the choice bits and correlation offsets according to two values that we are multiplying. However, in this case, we only do the oblivious transfer in one direction. We do not give the full protocol description as it is not easily presentable, and involves a lot of index manipulation for the general case with arbitrary L_i values.

We have also implemented this version and show the optimal L_i values for this strategy below.

Note that here, an L_i value of 2 means the receiver chooses from a total of $4^2 = 16$ messages, as we compute two bits from both multiplications in one go.

4.4.3 Summary

The total optimized communication in bits for different methods is presented in the table below.

Table 3: Optimal L_i values in terms of total communication when using 1-out-of-4^{L_i} strategy in triple generation.

Triple length ℓ	**L_i values**	**Required KK13 security parameter κ**	**Communication (bits)**
8	(2,2,2,2)	240	1440
16	(2,2,2,2,2,2,2,2)	240	3360
32	(1, ..., 1)	192	8064
64	(1, ..., 1)	192	19200

Table 4: Optimized communication in bits for single ℓ-bit triple computation with different methods.

Triple length ℓ	**baseline ALSZ13**	**KK13 1-out-of-2^{L_i}**	**KK13 1-out-of-4^{L_i}**
8	2176	1440	1440
16	4480	3360	3360
32	9472	7948	8064
64	20992	19200	19200

It can be seen that the advantage of using KK13 protocol over the baseline is larger for smaller bit-lengths. We also see that using KK13 with $\binom{4}{1}$-OT does not have communication gains compared to using $\binom{2}{1}$-OT. However, the advantage is that only one-way oblivious transfer is required and we do not have to bootstrap OT extension in both directions. However, round-complexity is not reduced in principle, since in other cases we can perform the OT-s in both directions in parallel.

It's also important to compare how much more computation the KK13 protocols require, in terms of calls to the hash function. The table below presents number of hash calls that a single computing party makes in total for computing one ℓ-bit triple.

Note that for the KK13 $\binom{4}{1}$-OT case, we have that one party has the sender role and the other the receiver. In the $\binom{2}{1}$-OT strategies, the two multiplications are done with switched roles, and therefore, both parties perform the same amount of hash calls. We can see that for smaller bit lengths (8 and 16), our KK13 protocols require a few more hash computations than ALSZ13, but the improvement in

Table 5: Total number of hash function calls for a single computing party when computing one ℓ-bit triple with the communication-optimized protocols.

Triple length ℓ	**baseline ALSZ13**	**KK13 1-out-of-2^{L_i}**	**KK13 1-out-of-4^{L_i} (sender/receiver)**
8	24	32	28 / 4
16	48	64	56 / 8
32	96	84	96 / 32
64	192	128	192 / 64

communication size is also larger for these lengths. For 32-bit and 64-bit triples, the KK13 protocols require the same amount or less hash calls, but also the communication improvement is less significant.

4.5 Implementation details

We have implemented both of the KK13 based triple generation protocols and also the ALSZ13 baseline protocol. Our implementation currently lacks a base OT, but the base OT has very little amortized effect on the overall precomputation performance. For example, the benchmarks of [DSZ15] report less than a second for performing the base OT-s. Our entire implementation is written in C++.

4.5.1 PRG

To instantiate the PRG, we use the AES-128 block cipher in CTR mode for the PRG, which gives us 128-bit security. In the case where the OT extension parameter $\kappa = 128$, we seed only the AES key with 128 bits, and take the IV (initialization vector) as 0. For larger κ values (in KK13 protocols), we use the extra bits to also seed the IV. This gives a unique PRG for each seed (supporting up to 256 bit seeds), while still retaining at least 128-bit security for the PRG, according to the NIST recommendations [BK12].

Using AES allows us to leverage the Intel AES-NI instruction set for much better performance than a software implementation of AES. Especially, we can process 8 blocks of output in parallel in CTR mode. We do not need to use AES-NI intrinsics explicitly in our implementation code, as we use the OpenSSL implementation of AES, which automatically uses AES-NI instructions by default if they are supported by the hardware. Also, parallel encryption/decryption is handled by the OpenSSL implementation in block cipher modes that allow it.

Overall, the PRG computations are not a bottleneck in our protocols, even though we need to generate pseudorandomness from κ different PRG-s at the same time.

4.5.2 Hash function

For the correlation-robust hash function (or random oracle) we used the SHA-256 function in all cases. Currently we consider the security of SHA-256 to be sufficiently well established to make it a suitable candidate for a strong hash function to use in our OT extension protocols. There is of course a gap between the correlation robustness definitions and an actual real-life hash function used in practical applications. However, the correlation robustness assumption is definitely weaker than that of the random oracle, as we know that random oracles do not exist. In practical applications, however, random oracles are also commonly instantiated with a single hash function.

Calculating the hash function was the clear bottleneck in the LAN (local area network) setting of our benchmarks, where the communication links are fast, taking up to 80% of the total running time. Due to this, we use multiple parallel threads for calculating the hashes and we performed benchmarks with different numbers of hashing threads. A SHA-256 implementation, which could leverage hardware SIMD (*single-instruction multiple-data*) instructions for calculating many hashes in parallel on a single thread would be very beneficial for increased performance. Currently, we use the OpenSSL implementation of SHA-256, as local benchmarks showed it is more efficient then the CryptoPP library one.

We also briefly considered and tested other instantiations, in particular SHA-3 and an improved version of one of the SHA-3 finalists, BLAKE2. For SHA-3, the only C++ implementation we found was from the CryptoPP library. Initial benchmarks showed that it was ~2 times slower than SHA-256.

For BLAKE2, we tested the official implementation (`https://github.com/BLAKE2/BLAKE2`). Specifically, we used the SIMD-optimized Blake2b variant with 32-byte outputs. Although the BLAKE2 official web-site advertises very high performance, we only observed relatively little performance gains in our protocols compared to OpenSSL's SHA-256. We suspect that the function and perhaps the implementation also is fine-tuned for computing a single hash from very large input data, but not for our use case of computing a huge amount of hashes on relatively small inputs.

For benchmark comparisons with the ABY implementation [DSZ15], we also implement hashing via fixed-key AES for the ALSZ13 protocol. The ABY paper presents performance results for generating triples with ALSZ13 protocol using the hash:

$$H(x,t) = AES_K(x \oplus t) \oplus x \oplus t.$$

for input x and monotonically increasing nonce t. For the ALSZ13 correlated-OT protocol, this construction of H needs to be a random oracle for security. The assumption that fixed-key AES is an ideal random permutation is not enough for H to be a random oracle. In fact, in the paper by Bellare et al [BHKR13], from where this construction is borrowed, it is explicitly stated that the construction is not indifferentiable from a random oracle. The [BHKR13] paper assumes AES is an ideal random permutation and then gives concrete security bounds when this construction is used in a larger protocol, specifically in Yao's garbled circuits protocol. Therefore, it seems dubious in terms of security to use this construction of H directly as a random oracle. As stated, we default in our implementation to SHA-256 as a better candidate for the strong security properties we require. However, due to AES-NI instructions, the above construction is of course much more efficient. Using multiple parallel threads for hashing evens out this overhead, but also requires more powerful hardware.

4.5.3 Bit-level operations

Our bit matrix transposition uses a sequential algorithm, which employs Intel's SIMD AVX2 instructions. AVX2 constructions allow to operate directly with 256-bit registers. In our case, we use these operations for bitwise XOR and bitwise AND and a few other specific operations. With AVX2 instructions, we can perform these bitwise operations on 256 bit inputs in roughly the same amount of processor cycles, when performing a single 64-bit bitwise operation.

Our bit matrix transposition implementation is based on the code from[12], modified to use AVX2 instructions, as the original code uses only SSE2 instructions with access to 128-bit registers. Bit matrix transposition is required in all OT extension protocols we have considered and is a rather costly computational task, as already noted in [ALSZ13].

Local tests show that Eklundh's algorithm [Ekl72] used in ABY (and originally proposed in [ALSZ13]) performs slightly better than AVX2-based sequential algorithm. Even more gains might be possible with a implementation of Eklundh's algorithm that leverages the SIMD-instructions. We currently have not implemented Eklundh's algorithm ourselves due to time constraints but this would also help optimize our implementation.

4.5.4 Batching

Our current implementation generates triples in large batches. After some initial testing, we chose to generate 100 000 triples in a single batch, so that the time it takes to complete one batch is not too large. To generate a million triples, we

[12]https://mischasan.wordpress.com/2011/10/03/the-full-sse2-bit-matrix-transpose-routine/

calculate ten of these batches sequentially and so on. Note that for generating 100 000 triples, the corresponding batch size for OT is much higher. E.g for the baseline ALSZ13 protocol and 64-bit triples, we have to perform 12 800 000 OT-s in total using the OT extension.

We observed that performing triple generation in smaller batches sequentially leads to longer total running time. However, doing the computation in one large batch means that there is a significant overhead introduced by one party having to wait for the other party to finish its computations and send the corresponding network message. This overhead is not related to bandwidth or latency, but rather the data dependencies of the network message on local computations. We tried to reduce this overhead by using parallel threads for hashing, which is the most intensive computation done in the protocols. However, using the PRG-s and transposing bit matrices also takes up noticeable time on very large matrices.

Overall, in hindsight we can say a much more efficient batching strategy would be to run smaller batches independently in parallel. This means local computations and network communication is naturally more interleaved, although some overhead is introduced for multiplexing the messages. Our current implementation does not support this, but is a solid optimization strategy for the future. We also believe it would give the KK13-based protocols more of an advantage over ALSZ13, even when using a slower hash function. The current benchmark results presented next support this hypothesis.

4.6 Benchmarks of precomputation techniques

We benchmarked triple generation based on three different OT extension flavors and with the optimizations discussed for reducing communication:

1. **ALSZ13** — $\binom{2}{1}$-COT extension from [ALSZ13]. One OT in both directions for each bit of the triple component.
2. **KK13-1-out-of-2** — $\binom{2^{L_i}}{1}$-COT OT extension from [KK13]. One OT in both directions, each OT computes L_i bits of the result.
3. **KK13-1-out-of-4** — $\binom{4^{L_i}}{1}$-COT OT extension from [KK13]. A single OT in one direction, each OT computes L_i bits of the result.

4.6.1 Hardware used for benchmarking

The benchmarks were performed on a cluster of two machines, with a dedicated fast 10 Gbit/s network link, 128 GB of RAM and two Intel Xeon E5-2640 v3 2,6 GHz/8GT/20M processors, meaning a total of 16 cores and 32 parallel threads with Intel HyperThreading.

4.6.2 Network conditions

We performed benchmarks in both a LAN and WAN (*wide-area network*) setting. The LAN setting means that network performance of the communication channel is very high, especially, latency is very low. The WAN setting simulates low performance network conditions, or when the computing parties are located very far from each other geographically. We simulate the WAN setting in our local cluster by using the Linux command line tool `tc` (traffic control). The WAN setting parameters were chosen to match the benchmark conditions of [DSZ15] to be able to compare them directly, since their precomputation implementation is the most recent and state-of-the-art available in the same security model as ours.

- LAN — ~ 0.1ms latency and up to 10Gbit/s bandwidth
- WAN — 170 ms latency (round-trip time) and throttled bandwidth at 70 Mbit/s (peak rate at 100 Mbit/s)

4.6.3 Total triple generation time

Times are reported in seconds for computing a total of 100 000 triples in a single batch. The average time of ten iterations of these batches is shown for each experiment. We performed tests in both LAN and WAN settings and using either 1, 4 or 16 parallel hashing threads. We did not observe any significant performance gains when using more than 16 threads for batch size of 100 000 triples, probably due to thread creation and scheduling related overhead.

We also include the benchmark results from [DSZ15] for 128-bit security in our tables for comparison. Note that ABY benchmarks uses 2 parallel threads, that perform OT on smaller batches, but use the network link in parallel. As such, their batching strategy is more efficient then ours currently.

Also, as their implementation uses a much faster hash construction based on AES, as we described above, we benchmarked our own baseline ALSZ13 with the same hash construction for comparison (denoted as ALSZZ13 AES). The results are presented below in Table 6.

Table 6: Total running times in seconds for the triple generation to compute 100 000 triples with different OT extension methods.

Threads	**Network**	**Triple length**	**ALSZ13**	**ALSZ13 AES**	**KK13-$\binom{2}{1}$**	**KK13-$\binom{4}{1}$**	**ABY**
1	LAN	8	1.94	0.60	3.07	2.87	0.3
1	LAN	6	3.42	1.12	5.65	5.24	0.6
1	LAN	32	6.32	1.97	7.42	5.91	1.1
1	LAN	64	12.01	3.82	11.93	11.74	2.7
1	WAN	8	6.05	3.98	7.29	6.49	4.6
1	WAN	16	11.85	7.75	12.78	12.16	8.2
1	WAN	32	21.94	16.02	21.56	20.41	11.1
1	WAN	64	44.33	35.16	44.81	44.73	22.4
4	LAN	8	0.91	0.53	1.13	1.14	0.3
4	LAN	16	1.62	0.96	2.15	1.90	0.6
4	LAN	32	3.09	1.71	3.05	2.63	1.1
4	LAN	64	5.49	3.25	5.31	5.14	2.7
4	WAN	8	4.39	3.90	3.84	3.65	4.6
4	WAN	16	8.58	7.61	8.39	7.75	8.2
4	WAN	32	18.10	15.45	16.55	16.63	11.1
4	WAN	64	37.93	33.89	38.26	38.27	22.4
16	LAN	8	0.60	0.48	0.57	0.54	0.3
16	LAN	16	1.10	0.86	1.01	0.99	0.6
16	LAN	32	1.93	1.60	1.64	1.71	1.1
16	LAN	64	3.51	3.09	3.24	3.08	2.7
16	WAN	8	3.99	3.92	3.01	2.89	4.6
16	WAN	16	7.75	7.53	6.32	6.20	8.2
16	WAN	32	15.95	15.52	14.11	14.80	11.1
16	WAN	64	34.86	34.07	34.94	35.95	22.4

Main observations

- ALSZ13 with fixed-key AES is fastest in all cases where 1 or 4 hashing threads are used (exception is 8-bit triples in WAN setting). This is because a single thread can produce 8 hashes in SIMD-parallel manner using AES-NI which is much faster than non-SIMD software implementation of SHA-256. We can also see that the ALSZ13-AES does not gain much performance from using multiple threads, since the hashing is fast as it is.
- With 16 hashing threads, this difference is evened out, and network becomes more of a bottleneck for all protocols, giving the KK13 protocols a competitive edge due to reduced communication, even in the LAN setting.

- The KK13 protocols perform favorably in most cases for 32 and 64-bit triples, compared to ALSZ13 using the slower SHA-256 hashing. For 8 and 16-bit triples, the KK13 are faster when using many hashing threads, since they require more calls to the hash function than ALSZ13.
- Although the KK13 perform mostly better in WAN setting than ALSZ13, an interesting exception is 64-bit triples, where the KK13 protocols are faster in the LAN setting but actually slower in the WAN setting compared to ALSZ13. This can only be explained by the unoptimal batching strategy and scheduling of local computations and network communication, since for 64-bit triples, the KK13 protocols need to do less hashing and less communication in total.
- Our implementation of ALSZ13-AES comes close but does not perform as well as benchmark results in ABY paper (except 8 and 16-bit triples in WAN setting). We suspect this is due to the current unoptimal batching strategy, since ABY performs the two-directional OT-s in parallel in two threads.
- Comparing KK13 1-out-of-2 and 1-out-of-4 methods, we can see that for 8 and 16-bit triples, the 1-out-of-4 strategy is the winner. For 32-bit triples, the 1-out-of-2 strategy requires slightly less hashing and also has slightly lower total communication. For 64-bit triples, the 1-out-of-2 strategy needs to perform 2/3 of the hashing done by 1-out-of-4 protocol. For these reasons, 1-out-of-2 performs slightly better with 32 and 64-bit triples. Arguably, when the two-directional OT-s were to be performed in parallel, the 1-out-of-2 strategy should win more from this in terms of efficiency.

4.6.4 Breakdown of different operations

Here we show the relative cost of different operations performed in the OT extension protocols from the total time of triple generation. On a high level, the total computation of the triple generation protocol of Protocol 6 is divided into two phases for a single computation party, a sender OT routine and a receiver OT routine, denoted as “ot_sender” and “ot_receiver”. The exception is when using the KK13 protocol based on 1-out-of-4 OT, as there, one party only performs the sender routine, and the other party acts as the receiver.

In the OT routines, we measured the cost of different local operations and also the time spent on waiting for network messages (send is non-blocking so we can measure the network overhead only from the receiving end). The different measured operations are:

1. **PRG** — time spent on generating pseudo-randomness. In all cases, the OT sender routine generates twice as much pseudo-randomness.

2. **transpose** — time spent on performing bit matrix transpositions. For ALSZ13, both parties perform a single transposition. For KK13, the sender performs two transpositions and the receiver performs one. The order of operations is also slightly different from ALSZ13.

3. **hashing** — time spent on hashing and building the masked matrices. Since we use correlated-OT in all cases, for 1-out-of-N OT, the sender calculates (N-1) hashes and the receiver calculates one hash for each transferred message.

4. **receive** — time spent on waiting for network messages.

Main observations

- In LAN settings, with small number of hashing threads, computing the hashes is the clear bottleneck (specifically, for OT sender), taking up to 93% of the total time for KK13 1-out-of-4 protocol with one thread and 8-bit triples.

- In most cases, the transpose operation took more than twice the amount of time than computing pseudorandomness with the PRG-s and was up to 20% of the whole computation time. In the ALSZ13 protocol with fixed-key AES hashing, the transpose operation in fact took more time than hashing (20% vs 13% for 16 threads in LAN). For the KK13 protocols, the transpose operation is also more costly compared to ALSZ13 because of the larger security parameter and length of the matrix rows.

- For ALSZ13 with fixed-key AES hashing, hashing is not a bottleneck, and already with one thread, the network is the main bottleneck.

- In the WAN setting, the network is the clear bottleneck for all protocols, which is to be expected, given the relatively low network performance for the simulated WAN.

- For the KK13 protocols with 16 threads in both network settings, we can see that the relative proportion of the receive operation is lower than for ALSZ13 protocol in the OT sender routine. The difference is higher for smaller bit-length triples, which exactly corresponds to the relative difference in total communication required for these protocols. Therefore, it is clear that the KK13 protocols get a performance gain on the account of needing to do less communication.

- In the receiver phase, the proportion of the receive operation in KK13 protocols is roughly the same or even higher when using 16 threads than the

ALSZ13 protocol. This can be explained by the unoptimal batching and network scheduling in the current implementation. Proportionally, the sender spends more time hashing, and therefore, the receiver proportionally waits longer for the matrix rows masked with the hash output to arrive.

- These findings suggest that the KK13 protocols especially could gain a lot in terms of performance when implementing a better batching strategy, e.g running OT in smaller batches, but in parallel. This would decrease the network scheduling overhead and would most likely eliminate the anomaly of KK13 protocols being slower for 64-bit triples in WAN setting.

We depict in Table 7 and Table 8 below only the relative cost of the “receive” operation, as the difference time spent on computation vs communication says the most about the performance profiles of protocols. These tables support most of the observations made above.

4.7 Conclusions on the precomputation benchmarks

Overall, the process of implementing the protocols described in this thesis, together with the various optimizations, showed that there is a considerable amount of engineering effort required to get the desired performance results. In particular:

- Hashing is the most critical local operation in terms of total performance, especially when using a standard hash function that is widely considered to be suitable for instantiating constructions that require strong security properties, such as SHA-256. Either SIMD- or thread-parallel (or both) hashing is required for good performance.

- Optimal scheduling of local computations and network communication between the parties is also crucial for performance, and the current implementation is lacking in this aspect. A good solution would be to run oblivious transfers in relatively small batches but in parallel to minimize the scheduling overhead as is the general approach in [DSZ15].

- As was already brought out in [ALSZ13], we confirmed that bit matrix transposition also takes a significant proportion of local computation time. Using Eklundh’s algorithm as proposed in [ALSZ13] could improve our implementation as well, and might additionally be enhanced with hardware-accelerated SIMD-instructions.

We believe that implementing a better batching strategy and more efficient bit matrix transposition would lead to an overall performance improvement to all

Table 7: Average percentage from OT sender routine total time spent on network communication (blocking receive).

Threads	**Network**	**Triple length**	**ALSZ13**	**ALSZ13 AES**	**KK13-$\binom{2}{1}$**	**KK13-$\binom{4}{1}$**
1	LAN	8	11.8	42.4	3.1	1.8
1	LAN	16	13.0	42.8	3.3	1.6
1	LAN	32	12.8	42.0	5.8	7.5
1	LAN	64	12.1	40.7	9.3	6.0
1	WAN	8	68.6	91.4	37.9	44.3
1	WAN	16	69.5	91.9	38.1	41.2
1	WAN	32	70.7	92.5	54.0	66.1
1	WAN	64	71.6	92.8	65.1	65.8
4	LAN	8	25.7	49.0	8.3	5.0
4	LAN	16	26.3	48.8	9.0	3.9
4	LAN	32	25.8	50.6	14.9	17.8
4	LAN	64	25.8	47.7	22.5	16.4
4	WAN	8	84.8	93.4	63.7	63.8
4	WAN	16	85.7	94.0	63.9	66.8
4	WAN	32	85.3	94.5	76.8	84.3
4	WAN	64	87.4	94.7	83.6	84.5
16	LAN	8	37.6	53.9	19.1	11.1
16	LAN	16	41.3	54.6	18.9	11.0
16	LAN	32	41.7	53.0	29.3	34.5
16	LAN	64	42.3	52.6	37.7	28.8
16	WAN	8	91.0	94.4	80.1	80.6
16	WAN	16	92.0	94.6	81.4	81.1
16	WAN	32	92.6	95.2	88.5	91.6
16	WAN	64	93.1	95.3	91.3	91.8

Table 8: Average percentage from OT receiver routine total time spent on network communication (blocking receive).

Threads	**Network**	**Triple length**	**ALSZ13**	**ALSZ13 AES**	**KK13-$\binom{2}{1}$**	**KK13-$\binom{4}{1}$**
1	LAN	8	44.9	38.6	83.6	84.4
1	LAN	16	44.3	41.8	85.1	84.6
1	LAN	32	44.7	38.2	77.1	62.8
1	LAN	64	46.4	41.8	65.0	66.8
1	WAN	8	82.1	91.1	93.0	93.3
1	WAN	16	83.8	92.1	93.4	93.3
1	WAN	32	84.3	92.6	91.9	89.5
1	WAN	64	85.7	93.8	90.6	91.5
4	LAN	8	39.9	41.9	71.6	72.9
4	LAN	16	41.7	44.8	73.9	71.3
4	LAN	32	43.2	40.9	66.6	52.3
4	LAN	64	42.1	41.6	55.1	55.8
4	WAN	8	87.6	92.7	91.5	92.1
4	WAN	16	89.2	93.3	93.1	93.0
4	WAN	32	90.6	93.5	93.4	92.6
4	WAN	64	91.6	94.5	93.6	94.2
16	LAN	8	43.9	40.5	56.2	54.4
16	LAN	16	43.2	43.7	57.3	50.9
16	LAN	32	42.8	42.5	52.5	42.3
16	LAN	64	45.3	43.2	47.8	47.2
16	WAN	8	91.1	92.8	91.3	91.9
16	WAN	16	91.9	93.6	92.9	92.6
16	WAN	32	93.0	94.1	93.9	93.5
16	WAN	64	94.0	95.0	94.6	95.4

tested versions of the triple generation protocol and also give a more noticeable performance gain to the KK13-based versions. We can use the existing Sharemind's network layer multiplexing capability to perform many batches in parallel to better utilize the network.

Although our implementation is currently lacking in some aspects that makes local computations still a partial bottleneck, we can already see concrete performance gains from using the KK13-based protocol with reduced total communication. The optimizations that reduce communication cost can also be further improved. In particular, the message sizes used in OT can be brought down even more, by implementing bit-level packing. Currently, we send full bytes always and as such, do not gain improvement from message sizes that are not byte-aligned.

Additionally, lower security parameter values can be used when doing OT-s with different number of messages in a batch. Currently we use the maximum required security parameter for the whole batch. However, it is not entirely, clear, whether the reduced communication cost would overweigh the added complexity in computation. Using both of these optimizations would also change the optimal L_i values in the OT message scheduling for KK13-based versions.

5 A protocol suite for two-party computation

In this section, we now to turn to the online phase of our implemented two-party protocol suite. As a result of this work, we have implemented the `shared2p` protection domain on Sharemind[13]. This section gives a full specification of the operations available in `shared2p`. We implemented `shared2p` from the ground up, although relying on the general code structure already present for other protection domains and the general Sharemind framework.

We mostly follow the design of Sharemind's existing three-party protocol suite as our ultimate goal is to provide an equivalent alternative to the three-party scenario, both in terms of performance and security, as much as that is possible. The three-party scenario allows for an efficient information-theoretically secure approach, the practical security of which can be based entirely on cryptographically secure random number generation[14].

In the two-party case, achieving information-theoretic security is unfortunately impossible [BGW88]. Therefore, we have to rely on computational hardness assumptions for security. Furthermore, the two-party scenario adds the entire complexity of the precomputation phase, which is entirely unnecessary in the three-party case. We have already seen that some computational hardness assumptions were necessary for securing the precomputation phase. In fact, the online phase does not add any additional requirements in this respect for security the BGW-style protocols that we present here.

The protocols presented here are adaptations of the existing three-party protocols into the two-party setting [BNTW12, LR15, KLR16]. In some cases, the two-party setting allows for some algorithmic optimization compared to the three-party protocol. The three-party protocols often use a temporary resharing of a value between two of the three parties, such that the third party's share is 0. In the two-party case, we naturally have this sharing and no further resharing is required.

For completeness, we provide the full descriptions of all protocols. We also show the input privacy for the multiplication protocol (and the analogous conjunction protocol), which implies the input privacy of all the rest of the protocols through composition. In fact, all protocols we present here are a composition of local oper-

[13]Sharemind's existing three-party protocol suite is similarly named `shared3p`, referring to the secret-shared representation of confidential data, and the requirement of three computing parties.

[14]Additionally, the implicit assumption of secure authenticated channels is required. This assumption is very common in SMC literature, as network channels can be secured in practice with standard methods of public key cryptography. Especially, the TLS (*transport layer security*) protocol is used in practice to ensure that tampering or eavesdropping on the channels is not possible for the adversary.

ations and the Beaver triple multiplication and conjunction protocols. Therefore, the security of the online phase is based purely on the security of precomputation.

All protocols described in this section are carried out between two computing parties $\mathcal{CP}_1$ and $\mathcal{CP}_2$. Whenever we refer to computing parties $\mathcal{CP}_i$ and $\mathcal{CP}_j$, we implicitly mean different computing parties, that is $i \neq j$.

5.1 Data representations

We first start by describing the fundamental data types and their secret-shared representation available for programming with SecreC in the `shared2p` protection domain. As in Sharemind's three-party protection domain, we use both additively and bitwise shared data types. On additively shared integers, we can build protocols that rely on precomputed multiplication triples.

For bitwise shared data types, we use conjunction triples to build BGW-style protocols. We can also add a floating-point arithmetic suite for bitwise shared IEEE 754 numbers by relying on the Yao's garbled circuits protocol and methods presented in [PS15]. Although we have currently not implemented this in the `shared2p` protection domain, the protocols can be imported from the three-party protocol suite in a straightforward manner. We outline the necessary techniques in Section 5.8.

The paradigm of mixing different methods of secure computation for better efficiency has been used in previous works as well, including in protocols for Sharemind's three-party protection domain that switch between the additive and bitwise representations [BNTW12]. Most recently, the ABY framework uses both additive and bitwise shared representation and also includes Yao's protocol in addition to BGW-style protocols based on Beaver triples [DSZ15].

5.1.1 Additively shared data types

It is very natural to represent signed and unsigned integers of fixed bit-length using additive sharing. Unsigned integers with bit-length ℓ are exactly represented with elements of the ring $\mathbb{Z}_{2^\ell}$.

For signed integers we use two's complement notation, which is also the standard representation in regular computers. In two's complement, an ℓ-bit signed integer is represented also as an integer $a \in \mathbb{Z}_{2^\ell}$, but it's value is interpreted as

$$v = -a[\ell] \cdot 2^{\ell-1} + \sum_{i=1}^{\ell-1} a[i] \cdot 2^{i-1} \quad .$$

Essentially, the most-significant bit defines the sign $(+/-)$ of the integer, and the remaining $\ell-1$ bits define its value. The two's complement is a popular representation for signed integers, since addition, subtraction and multiplication use exactly the same operations as for unsigned integers. Therefore, we can conveniently use an additively shared representation over $\mathbb{Z}_{2^\ell}$ for signed integers as well.

A summary of additive data types is presented in Table 9.

Table 9: Additively shared data types in the `shared2p` protection domain.

Data type	**Domain of values**	**Secret-shared representation**	**Description**
`uint8`	Integers in $\mathbb{Z}_{2^8}$	Additive sharing over $\mathbb{Z}_{2^8}$	8-bit unsigned integer
`uint16`	Integers in $\mathbb{Z}_{2^{16}}$	Additive sharing over $\mathbb{Z}_{2^{16}}$	16-bit unsigned integer
`uint32`	Integers in $\mathbb{Z}_{2^{32}}$	Additive sharing over $\mathbb{Z}_{2^{32}}$	32-bit unsigned integer
`uint64`	Integers in $\mathbb{Z}_{2^{64}}$	Additive sharing over $\mathbb{Z}_{2^{64}}$	64-bit unsigned integer
`int8`	$[-2^7, 2^7 - 1]$, represented as elements of $\mathbb{Z}_{2^8}$ in two's complement notation	Additive sharing over $\mathbb{Z}_{2^8}$	8-bit signed integer
`int16`	$[-2^{15}, 2^{15} - 1]$, represented as elements of $\mathbb{Z}_{2^{16}}$ in two's complement notation	Additive sharing over $\mathbb{Z}_{2^{16}}$	16-bit signed integer
`int32`	$[-2^{31}, 2^{31} - 1]$, represented as elements of $\mathbb{Z}_{2^{32}}$ in two's complement notation	Additive sharing over $\mathbb{Z}_{2^{32}}$	32-bit signed integer
`int64`	$[-2^{63}, 2^{63} - 1]$, represented as elements of $\mathbb{Z}_{2^{64}}$ in two's complement notation	Additive sharing over $\mathbb{Z}_{2^{64}}$	64-bit signed integer

5.1.2 Bitwise shared data types

The additive representation is not well-suited for efficiently computing operations that intrinsically require access to the bitwise representation of the integer. A prime example among the protocols implemented here are comparison protocols, which rely on temporary bitwise shared representation inside the protocol. We thus provide also a separate integer data type that is bitwise shared.

We arbitrarily present the boolean data type also as a bitwise one, since

Booleans are associated with bitwise operations, although for elements of $\mathbb{Z}_2$, the additive and bitwise secret sharing scheme are equivalent. Also, we use conjunction triples for multiplying Booleans, since they are more efficient to compute than arithmetic multiplication triples. A summary of bitwise shared data types is presented in Table 10.

Table 10: Bitwise shared data types in the `shared2p` protection domain.

Data type	**Domain of values**	**Secret-shared representation**	**Description**
`bool`	Boolean values in $\mathbb{Z}_2$	Additive/bitwise sharing over $\mathbb{Z}_2$	Boolean value
`xor_uint8`	Integers in $\mathbb{Z}_{2^8}$	Bitwise sharing over $\mathbb{Z}_2^8$	8-bit unsigned integer
`xor_uint16`	Integers in $\mathbb{Z}_{2^{16}}$	Bitwise sharing over $\mathbb{Z}_2^{16}$	16-bit unsigned integer
`xor_uint32`	Integers in $\mathbb{Z}_{2^{32}}$	Bitwise sharing over $\mathbb{Z}_2^{32}$	32-bit unsigned integer
`xor_uint64`	Integers in $\mathbb{Z}_{2^{64}}$	Bitwise sharing over $\mathbb{Z}_2^{64}$	64-bit unsigned integer

5.2 Classify, declassify and publish

We now describe the fundamental protocols that transform public values to private values and vice versa, which form a basis for manipulating with confidential data in the protection domain. For example, we need a protocol to securely share the private input of an input party between the computing parties. We call this protocol Classify. We separately consider the case when an input party is at the same time also a computing party.

Similarly, the Declassify protocol publishes a secret-shared value to a party. With Declassify, we specifically refer to making the value public for all computing parties. In the case where the computing parties reveal a secret value to an external result party, we use a Publish protocol.

Additionally, to end a composition of input-private protocols securely, we make use of the Reshare protocol presented already with our Beaver triple generation protocol in Section 4. Note that Classify, Declassify and Publish protocols presented here are identical for both additive and bitwise types. For simplicity, we consider only the additive case in security proofs.

Recall that we use $\mathcal{F}(\cdot, \ldots, \cdot) = [\![y]\!]$ to denote an ideal functionality that explicitly outputs a random sharing of the value y.

5.2.1 Classify

A trivial way to classify a value x known to one of the computing parties, is simply to do nothing. The other party sets its share to 0, which constitutes a valid sharing of x, albeit not a secure one. However, the protocol in Protocol 10 is still trivially correct and input-private and can therefore be composed with Reshare for a secure protocol, or alternatively be composed with other input-private protocols.

Protocol 10 Classify of ℓ-bit types

Functionality: $\mathcal{F}(x, \perp) = [\![x]\!]$
Input: $\mathcal{CP}_i$ holds a value $x \in \mathbb{Z}_{2^\ell}$
Result: Shared value $[\![x]\!]$ between $\mathcal{CP}_1$ and $\mathcal{CP}_2$

```
1: $\mathcal{CP}_i$ fixes $[\![x]\!]_i = x$
2: $\mathcal{CP}_j$ fixes $[\![x]\!]_j = 0$
3: return $[\![x]\!]$
```

Theorem 16. *The* Classify *protocol in Protocol 10 is perfectly input-private.*

Proof. The protocol is input-private, since no communication occurs and there are no messages to simulate. □

When considering Classify for an input party, we want the ideal functionality to provide a random sharing of the input to the computing parties, so neither of them learns the secret input. The protocol is thus straightforward to construct securely by using Share to produce a secure resharing, and then distributing the shares among the computing parties.

Protocol 11 Classify of ℓ-bit types for an input party

Functionality: $\mathcal{F}(x, \perp, \perp) = (\perp, [\![x]\!])$
Input: $\mathcal{IP}$ holds a value $x \in \mathbb{Z}_{2^\ell}$
Result: Shared value $[\![x]\!]$ between $\mathcal{CP}_1$ and $\mathcal{CP}_2$

```
1: $\mathcal{IP}$ computes shares $(x_1, x_2) \leftarrow \mathsf{Share}(x)$ and sends $x_i$ to $\mathcal{CP}_i$ for $i \in \{1, 2\}$
2: Both $\mathcal{CP}_i$ fix $[\![x]\!]_i = x_i$
3: return $[\![x]\!]$
```

Theorem 17. *The* Classify *protocol in Protocol 11 is perfectly secure.*

Proof. For corrupt $\mathcal{CP}_i$, we can simulate the message x_i perfectly by sending the ideal functionality's output $[\![x]\!]_i$ to the adversary. □

5.2.2 Declassify

Protocol 12 Declassify of ℓ-bit types

```
Functionality: $\mathcal{F}([\![x]\!]) = (x, x)$
Input: Shared value $[\![x]\!]$
Result: $\mathcal{CP}_1$ and $\mathcal{CP}_2$ learn $x$
1: Both $\mathcal{CP}_i$ send $[\![x]\!]_i$ to $\mathcal{CP}_j$
2: Both $\mathcal{CP}_i$ compute $x \leftarrow \mathsf{Combine}([\![x]\!]_1, [\![x]\!]_2)$
3: return $x$
```

Theorem 18. *The declassify protocol in Protocol 12 is perfectly secure.*

Proof. The protocol is trivially secure, as the simulator for corrupted $\mathcal{CP}_i$ receives the output x from the ideal functionality and can simulate the message $[\![x]\!]_j$ by computing $x - [\![x]\!]_i$. The simulation is perfect, since $x = [\![x]\!]_1 + [\![x]\!]_2$ by definition. □

Since the declassify protocol is secure, we can use it as a final step to end a composition of input-private protocols, to make the whole composition secure. In fact, no resharing step is required as opposed to the three-party setting, where the similar Declassify protocol is not secure without the Reshare protocol [Bog13].

5.2.3 Publish

When publishing the computation outcome to result parties, we still have to explicitly reshare the output before sending the shares to the result party. Otherwise, the result party could infer more information from the values of the individual shares then just their combined value. This may happen if the shares are output by an input-private protocol and not a secure one. Thus, when the final result of a computation is published to an external result party, we always need to end a composition of input-private protocols with the secure resharing protocol.

Protocol 13 Publish of ℓ-bit types to a result party

```
Functionality: $\mathcal{F}([\![x]\!], \perp) = (\perp, \perp, x)$
Input: Shared value $[\![x]\!]$
Result: Result party $\mathcal{RP}$ learns $x$
1: Compute $[\![x']\!] \leftarrow \mathsf{Reshare}([\![x]\!])$
2: Both $\mathcal{CP}_i$ send $[\![x']\!]_i$ to $\mathcal{RP}$
3: $\mathcal{RP}$ computes $x \leftarrow \mathsf{Combine}([\![x']\!]_1, [\![x']\!]_2)$
4: return $x$
```

Theorem 19. *The* Publish *protocol in Protocol 13 is perfectly secure.*

Proof. Security with respect to corrupt $\mathcal{CP}_i$ is trivial, since the parties have no output, and the only communication is performed during the perfectly secure Reshare protocol, which we can simulate.

For corrupt $\mathcal{RP}$, we can also perfectly simulate the shares $[\![x']\!]_1$ and $[\![x']\!]_2$ by first generating uniformly random $[\![\overline{x'}]\!]_1 \leftarrow \mathbb{Z}_{2^\ell}$ and then taking $[\![\overline{x'}]\!]_2 = x - [\![\overline{x'}]\!]_1$, where x is the ideal functionality's output for $\mathcal{RP}$. The simulation is perfect, since in the real protocol, the shares of $[\![x']\!]$ are also guaranteed to be uniformly random, since Reshare is perfectly secure. Note that the values of $[\![\overline{x'}]\!]_i$ and $[\![x']\!]_i$ do not have to match exactly, since the shares of $[\![x']\!]$ are not output by the ideal functionality and the environment cannot directly compare them. Therefore, we only need to make sure they are of the same distribution, which we have shown, since both are uniformly random. $\square$

5.3 Arithmetic protocols

5.3.1 Addition, subtraction and multiplication with constant

Since additive secret sharing is homomorphic in terms of addition, we can add and subtract additively shared types using only local operations. Also, multiplication with a public constant is communication-free due to distributivity. Let us fix additively shared values $[\![x]\!]$ and $[\![y]\!]$ over $\mathbb{Z}_{2^\ell}$.

Addition ($\mathcal{F}([\![x]\!], [\![y]\!]) = [\![z]\!]$, $z = x + y$). Both $\mathcal{CP}_i$ locally compute $[\![z]\!]_i = [\![x]\!]_i + [\![y]\!]_i$.

Subtraction ($\mathcal{F}([\![x]\!], [\![y]\!]) = [\![z]\!]$, $z = x - y$). Both $\mathcal{CP}_i$ locally compute $[\![z]\!]_i = [\![x]\!]_i - [\![y]\!]_i$.

Negation ($\mathcal{F}([\![x]\!]) = [\![z]\!]$, $z = -x$). Both $\mathcal{CP}_i$ locally compute $[\![z]\!]_i = -[\![x]\!]_i$.

Multiplication with public constant ($\mathcal{F}([\![x]\!], c) = [\![z]\!]$, $z = cx$). Given a public constant $c \in \mathbb{Z}_{2^\ell}$ known by both parties, both $\mathcal{CP}_i$ locally compute $[\![z]\!]_i = c \cdot [\![x]\!]_i$.

Correctness for all three protocols follows directly from the properties of additive sharing. Also, all protocols are trivially input-private, since no communication is performed between the computing parties. However, they are not secure w.r.t our ideal functionality, since they do not provide fresh random output shares.

5.3.2 Multiplication

For multiplying additive types, we use the standard Beaver triple multiplication protocol of [Bea91] presented as Protocol 14.

Protocol 14 Multiplication protocol of ℓ-bit additive types

Functionality: $\mathcal{F}([\![x]\!], [\![y]\!]) = [\![w]\!]$ such that $w = xy$
Setup: Parties $\mathcal{CP}_1$, $\mathcal{CP}_2$ share a Beaver triple $[\![a]\!] \cdot [\![b]\!] = [\![c]\!]$
Input: Secret-shared values $[\![x]\!], [\![y]\!]$
Result: Secret-shared value $[\![w]\!] = [\![x]\!] \cdot [\![y]\!]$

```
1: The parties compute $[\![e]\!] = [\![x]\!] - [\![a]\!]$ and $[\![d]\!] = [\![y]\!] - [\![b]\!]$
2: The parties reveal values $e$ and $d$ with Declassify($[\![e]\!]$) and Declassify($[\![d]\!]$)
3: Both $\mathcal{CP}_i$ compute $[\![w]\!]_i = [\![c]\!]_i + e \cdot [\![b]\!]_i + d \cdot [\![a]\!]_i$
4: $\mathcal{CP}_1$ computes $[\![w]\!]_1 = [\![w]\!]_1 + e \cdot d$
5: return $[\![w]\!]$
```

Theorem 20. *The Beaver triple multiplication protocol in Protocol 14 is correct.*

Proof. Due to distributivity, we have $e \cdot [\![b]\!]_1 + e \cdot [\![b]\!]_2 = eb$ and $d \cdot [\![a]\!]_1 + d \cdot [\![a]\!]_2 = da$. The correctness can then be shown with simple algebra:

$$\begin{aligned} w &= c + eb + da + ed = ab + (x-a)b + (y-b)a + (x-a)(y-b) \\ &= ab - xb - ab + ya - ab + xy + xb - ya + ab = xy \ . \end{aligned}$$

□

In the passive two-party setting, this is the most efficient protocol for multiplication at least for the online phase. Due to the two's complement representation, the multiplication protocol works as intended also for signed integers. Each multiplication consumes a single multiplication triple of bit-length equal or larger than the arguments. Although we could use 64-bit multiplication triples for all data types, generating smaller bit-length triples is much more efficient as shown in Section 4.6 and therefore, we precompute separate triples for all required bit-lengths.

Based on the definition of security for a triple generation protocol (Def. 13), we can prove that the multiplication protocol in Protocol 14 is input-private.

Theorem 21. *Given that the Beaver triple used in Protocol 14 is output by a secure Beaver triple generation protocol according to Def. 13, the protocol in Protocol 14 is computationally input-private.*

Proof. For corrupt $\mathcal{CP}_i$, we need to simulate the shares $[\![e]\!]_j$ and $[\![d]\!]_j$ sent in the Declassify protocol. Since we use a secure Beaver triple generation protocol, the shares $[\![a]\!]_j$ and $[\![b]\!]_j$ are computationally indistinguishable from independently generated uniformly random values. Therefore, also $[\![e]\!]_j = [\![x]\!]_j - [\![a]\!]_j$ and $[\![d]\!]_j = [\![y]\!]_j - [\![b]\!]_j$ are uniformly random and we can simulate these values by generating new uniformly random values, which makes the resulting simulated protocol view computationally indistinguishable from a real execution. $\square$

To show security for multiplication, we would have to simulate the values of e and d such that the output of the simulated view exactly matches that of the ideal functionality, given access to the ideal functionality's output. However, this does not seem to possible for the multiplication protocol. A secure version would therefore require a Reshare protocol at the end.

5.4 Protocols for bitwise operations

5.4.1 Bitwise XOR and conjunction with public constant

Bitwise operations are naturally more efficient in the bitwise sharing representation. Similarly to addition in additive sharing, the XOR operation can be computed locally on bitwise shared values. Let us fix bitwise shared values $[\![x]\!]$ and $[\![y]\!]$ over $\mathbb{Z}_2^\ell$.

XOR ($\mathcal{F}([\![x]\!],[\![y]\!]) = [\![z]\!]$, $z = x \oplus y$). Both $\mathcal{CP}_i$ locally compute $[\![z]\!]_i = [\![x]\!]_i \oplus [\![y]\!]_i$.

Bitwise logical negation ($\mathcal{F}([\![x]\!]) = [\![z]\!]$, $z = \neg x$) $\mathcal{CP}_1$ locally computes $[\![z]\!]_1 = \neg[\![x]\!]_1$ (flipping all bits in the shared value).

Conjunction with public constant ($\mathcal{F}([\![x]\!], c) = [\![z]\!]$, $z = x \wedge c$). Given a public constant $c \in \mathbb{Z}_2^\ell$ known by both parties, both $\mathcal{CP}_i$ locally compute $[\![z]\!]_i = c \wedge [\![x]\!]_i$.

Public bitshift left/right ($\mathcal{F}([\![x]\!], c) = [\![z]\!]$, $z = x \ll c$ or $z = x \gg c$). Given a public constant $c \in \mathbb{Z}_{2^\ell}$ known by both parties, both $\mathcal{CP}_i$ locally compute $[\![z]\!]_i = [\![x]\!]_i \ll c$. Similarly for $[\![z]\!] = [\![x]\!] \gg c$.

Correctness for these protocols follows from the properties of bitwise sharing and the protocols are trivially input-private, but not secure, since they do not produce fresh random shares.

5.4.2 Bitwise conjunction

For bitwise conjunction, we can use exactly the same protocol as for multiplication (Protocol 14), but relying instead on conjunction triples. Also, local additions and multiplications are replace with bitwise XOR and conjunction operations respectively. Input privacy follows analogously as in Theorem 5.3.2 if we use the secure conjunction triple generation protocol of Protocol 8.

Conjunction triples are also more convenient to precompute than multiplication triples, since we can combine arbitrary bit-length conjunction triples from bit triples, simply by concatenating the bits in the bitwise representation. This can be easily seen by considering that each bit taken separately in a bitwise shared conjunction triple is a 1-bit conjunction triple on its own.

5.4.3 Bitwise disjunction

For disjunction, we can use the simple equivalence with conjunction $x \vee y = \neg(\neg x \wedge \neg y)$.

Protocol 15 Bitwise disjunction protocol of ℓ-bit bitwise types

Functionality: $\mathcal{F}([\![x]\!], [\![y]\!]) = [\![z]\!]$ where $z = x \vee y$
Input: Bitwise shared values $[\![x]\!]$, $[\![y]\!]$
Result: Bitwise shared value $[\![z]\!]$

```
1: $[\![z]\!] = (\neg[\![x]\!]) \wedge (\neg[\![y]\!])$
2: return $[\![z]\!] = \neg[\![z]\!]$
```

The disjunction protocol of Protocol 15 is trivially correct and also input-private, given that the conjunction protocol is input-private.

5.5 Comparison protocols

We now present protocols for checking equality of and comparing secret-shared integers. For comparison protocols, we first have to define two specific sub-protocols for finding the most-significant non-zero bit, and checking whether shares of an additive integer overflow the modulus. We use these protocols also later for conversions between different data types. The overflow protocol also gives us a subroutine for extracting a single bit from an additively shared data type.

All protocols presented hereinafter are based either on [BNTW12] or from previously unpublished better optimized protocols existing in Sharemind's implementation of the three-party protection domain, that are written in either C++ or Sharemind's specialized protocol language [LR15] by the authors of [KLR16].

We adapt these protocols to the two-party case and avoid the steps where some values are temporarily shared between two parties in the three-party protocols.

Also, all following protocols are trivially input-private as a composition of input-private protocols.

5.5.1 Equality

The equality of two additively shared integers $[\![x]\!]$ and $[\![y]\!]$ is equivalent to the condition $[\![x]\!] - [\![y]\!] = 0$. To check if $[\![d]\!] = 0$ for additively shared d, we need to check whether $[\![d]\!]_1 = -[\![d]\!]_2$, which in turn is equivalent to $[\![d]\!]_1 \oplus (-[\![d]\!]_2) = 0$. Therefore, we can consider a bitwise shared number where $\mathcal{CP}_1$ takes $[\![d]\!]_1$ as his share, and $\mathcal{CP}_2$ takes $-[\![d]\!]_2$ and compute the disjunction of all bits of this number. This is 0 exactly if there are no non-zero bits, which happens exactly if $[\![d]\!]_1 = -[\![d]\!]_2$.

Protocol 16 Equality protocol of ℓ-bit additive types

Functionality: $\mathcal{F}([\![x]\!], [\![y]\!]) = [\![b]\!]_{\bmod 2}$, where $b = 1$ iff $x = y$
Input: Shared values $[\![x]\!]$, $[\![y]\!]$ over $\mathbb{Z}_{2^\ell}$
Result: Shared bit $[\![b]\!]_{\bmod 2}$

```
1: Initialize bitwise shared value $[\![d]\!]_{\oplus 2^\ell}$                ▷ $d = [\![x]\!] - [\![y]\!]$
2: $\mathcal{CP}_1$ computes $[\![d]\!]_1 = [\![x]\!]_1 - [\![y]\!]_1$
3: $\mathcal{CP}_2$ computes $[\![d]\!]_2 = -([\![x]\!]_2 - [\![y]\!]_2)$
4: Compute $[\![b]\!]_{\bmod 2} = \neg \left( \bigvee_{i=1}^{\ell} [\![d]\!][i] \right)$
5: return $[\![b]\!]$
```

Theorem 22. *The equality protocol for additive types in Protocol 16 is correct.*

Proof. We have that $b = 1$ exactly if all $d[i]$ are zero. This in turn means that $[\![x]\!]_1 - [\![y]\!]_1 = -[\![x]\!]_2 + [\![y]\!]_2$, from which it follows that $x = y$. □

Note that we can implement the disjunction of all bits of d more efficiently than a naive iterative algorithm, which would take $\ell - 1$ rounds. Instead, we can use a divide-and-conquer approach to first take the bitwise disjunction of the first half and the second half of the bits, which is a disjunction of $\ell/2$-bit values. Then continue the same process recursively until we are left with a single bit. This process takes exactly $\log_2 \ell$ rounds and we perform a total of $\ell - 1$ bit disjunctions.

Also, the equality protocol for bitwise types can be implemented analogously, but instead of considering the difference $x - y$, we consider the bitwise XOR $x \oplus y$. Performance and communication costs are the same as in the additive case.

5.5.2 Most significant non-zero bit

The most significant non-zero bit (MSNZB) protocol finds the position of the first non-zero bit of a bitwise shared integer, starting from the most significant one. For this, we use the protocol from [BNTW12]. The protocol returns a bitwise shared integer from the same domain, which has at most one non-zero bit at the position where the most significant non-zero bit in the argument is, or all zero bits if the argument is zero.

Protocol 17 MSNZB protocol of ℓ-bit bitwise types

Functionality: $\mathcal{F}(\llbracket x \rrbracket_{\oplus 2^\ell}) = \llbracket s \rrbracket_{\oplus 2^\ell}$ where $s[i] = 1$ for the largest value i, such that $x[i] = 1$ and $s[j] = 0$ for all $j \neq i$

Input: Bitwise shared value $\llbracket x \rrbracket$

Result: Bitwise shared value $\llbracket s \rrbracket$

```
1: Initialize bitwise shared value $\llbracket t \rrbracket = \llbracket x \rrbracket$
2: for $i \in \{1, \ldots, \log_2 \ell\}$ do
3:     Compute $\llbracket t \rrbracket = \llbracket t \rrbracket \vee (\llbracket t \rrbracket \gg 2^{i-1})$
4: end for
5: return $\llbracket s \rrbracket = \llbracket t \rrbracket \oplus (\llbracket t \rrbracket \gg 1)$
```

Theorem 23. *The* MSNZB *protocol in Protocol 17 is correct.*

Proof. Intuitively, in the for-cycle, we compute a so-called prefix-OR operation, which is defined as $t[i] = \bigvee_{j=i}^{j=\ell} t[j]$. That is, the i-th bit is the logical disjunction of itself and all bits in higher positions than i. We use a divide-and-conquer method to calculate the prefix-OR in $\log \ell$ rounds. Therefore, in the final result we have all non-zero bits starting from where x has the most significant non-zero bit. In the final row, we invert the trailing non-zero bits and keep only the most-significant non-zero bit in the result. □

Similarly to the equality protocol, we have a total of $\log_2 \ell$ rounds but we do more single bit disjunctions, namely $\ell \cdot \log_2 \ell$.

5.5.3 Overflow

The Overflow protocol presented in Protocol 18 checks whether the sum of shares for additively shared $\llbracket x \rrbracket \in \mathbb{Z}_{2^\ell}$ would overflow the modulus 2^ℓ. That is, whether $\llbracket x \rrbracket_1 + \llbracket x \rrbracket_2 \geq 2^\ell$ (without modular arithmetic). We actually present a more general protocol that allows also to specify the position where overflow should be checked. The protocol is based on [BNTW12].

Protocol 18 $\mathsf{Overflow}$ protocol of ℓ-bit additive types

```
Functionality: $\mathcal{F}([\![x]\!], t) = [\![b]\!]_{\bmod 2}$ where $b = 1$ iff $[\![x]\!]_1[1..t] + [\![x]\!]_2[1..t] \geq 2^t$
Input: Additively shared integer $[\![x]\!]$, public integer $t$ specifying overflow position
Result: Shared bit $[\![b]\!]$
1: Initialize bitwise shared value $[\![d]\!]_{\oplus 2^t}$    ▷ $d = ([\![x]\!]_1 \bmod 2^t) - ([\![x]\!]_2 \bmod 2^t)$
2: Both $\mathcal{CP}_i$ fix $[\![d]\!]_i = [\![x]\!]_i \bmod 2^t$
3: $\mathcal{CP}_2$ fixes $[\![d]\!]_2 = 0 - [\![d]\!]_2$
4: Compute $[\![s]\!] = \mathsf{MSNZB}([\![d]\!])$
5: $\mathcal{CP}_1$ fixes $[\![u]\!]_1 = 0$ and $\mathcal{CP}_2$ fixes $[\![u]\!]_2 = [\![d]\!]_2$    ▷ $u = (0 - [\![x]\!]_2) \bmod 2^t$
6: Compute $[\![t]\!] = [\![s]\!] \wedge [\![u]\!]$
7: Compute $[\![b]\!] = \neg \left(\bigoplus_{k=1}^{t} [\![t]\!][k]\right)$    ▷ XOR all bits of $[\![t]\!]$
8: If $[\![d]\!]_2 = 0$, $\mathcal{CP}_2$ computes $[\![b]\!]_2 = [\![b]\!]_2 \oplus 1$
9: return $[\![b]\!]$
```

Theorem 24. *The* $\mathsf{Overflow}$ *protocol in Protocol 18 is correct.*

Proof. For the correctness proof we assume $t = \ell$ for simplicity. The general case works the same way, by first converting the shares to $\mathbb{Z}_{2^t}$ by taking the modulus with 2^t.

The main idea of the protocol is that $[\![x]\!]_1 + [\![x]\!]_2 \geq 2^\ell$ is equivalent to $[\![x]\!]_1 \geq 2^\ell - [\![x]\!]_2$. In modulo 2^ℓ, we have that $2^\ell - [\![x]\!]_2 \equiv 0 - [\![x]\!]_2 \mod 2^\ell$, but for the case $[\![x]\!]_2 = 0$, we then get the wrong result, so this is treated separately. To make the comparison $[\![x]\!]_1 \geq 0 - [\![x]\!]_2$, we consider the bitwise shared representation of $[\![x]\!]_1 \oplus (0 - [\![x]\!]_2)$, which is computed as $[\![d]\!]$ in the protocol. We then use $[\![s]\!] = \mathsf{MSNZB}([\![d]\!])$ to determine the highest set bit of d. If there is a set bit, and $[\![d]\!]_2$ has a set bit at the same location, we have $[\![x]\!]_1 < 0 - [\![x]\!]_2$. This is checked first with the conjunction $[\![s]\!] \wedge [\![u]\!]$, where $[\![u]\!]$ is $0 - [\![x]\!]_2$ shared bitwise, and then XOR-ing all bits together to find if one of those was 1[15]. Since we negate the value of $[\![b]\!]$, we have that $b = 1$ exactly if $[\![x]\!]_1 \geq 0 - [\![x]\!]_2$. However, in the case $[\![x]\!]_2 = 0$, the inequality always holds, but there can be no overflow. We thus correct this by letting $\mathcal{CP}_2$ flip the value of $[\![b]\!]$ if $[\![x]\!]_2 = 0$. □

The round-complexity for overflow is that of MSNZB and bitwise conjunction added, which adds up to $\log_2 \ell + 1$. In total, we consume $\ell \cdot \log_2 \ell + \ell$ bit conjunction triples.

Note that the three-party protocol presented in [BNTW12] requires that the input is secret-shared between two parties, and the third party's share is 0. In the two-party case, this is naturally so, and we therefore do not have to explicitly reshare the input.

[15]Note that we can consider a bitwise shared integer as a vector of shared bits, and thus, XOR-ing together the bits is trivially done as a local operation.

5.5.4 Bit extraction

We now describe how to extract a single bit according to a known public position index from an additively shared integer $[\![x]\!]$ using the $\mathsf{Overflow}$ protocol. Intuitively, if we XOR together the k-th bits in the additive shares of x, we get the k-th bit in the actual value, unless the first $k-1$ bits overflow. We denote this sub-routine of extracting the k-th bit as $\mathsf{ExtractBit}([\![x]\!], k)$.

Protocol 19 $\mathsf{ExtractBit}$ protocol for extracting the k-th bit from ℓ-bit additive types

```
Functionality: $\mathcal{F}([\![x]\!], k) = [\![b]\!]_{\bmod 2}$ where $b = x[k]$
Input: Shared value $[\![x]\!]$ over $\mathbb{Z}_{2^\ell}$
Result: Shared bit $[\![b]\!]$
1: Compute $[\![b]\!]_{\bmod 2} = \mathsf{Overflow}([\![x]\!], k-1)$
2: Both $\mathcal{CP}_i$ compute $[\![b]\!]_i = [\![b]\!]_i \oplus [\![x]\!]_i[k]$
3: return $[\![b]\!]$
```

Theorem 25. *The* $\mathsf{ExtractBit}$ *protocol in Protocol 19 is correct.*

Proof. We first compute $[\![b]\!] = \mathsf{Overflow}([\![x]\!], k-1)$ to learn whether the addition of the shares of x propagates a carry bit into position k. Then, to extract the bit itself we additionally XOR this with the corresponding bits in the additive shares $[\![b]\!]_i = [\![b]\!]_i \oplus [\![x]\!]_i[k]$. This exactly matches the process of how the value k-th is determined when adding the shares of $[\![x]\!]$ together. □

The round-complexity and communication cost for $\mathsf{ExtractBit}$ is the same as for the $\mathsf{Overflow}$ protocol.

5.5.5 Less-than comparison

We have now described all the required sub-protocols for performing the less-than comparison on all types. Notice, that having a protocol for the less-than comparison is sufficient for performing also the less-than-or-equal comparison, since $x \leq y = \neg(y < x)$. Naturally, we can perform greater-than comparison also by switching the arguments.

Using the $\mathsf{ExtractBit}$ protocol, we can construct a comparison protocol for additive types. For unsigned integers, we perform the less-than comparison in Protocol 20 by comparing the most significant bits of the arguments, which we can compute with $\mathsf{ExtractBit}$. We can also easily modify the protocol for signed integers, as we do in Protocol 21. The protocol is not explicitly presented in previous publications, but follows the main ideas of [BNTW12] and is based on Sharemind's current optimized three-party protocol implementation.

Protocol 20 Less-than comparison protocol of ℓ-bit unsigned additive types

Functionality: $\mathcal{F}([\![x]\!], [\![y]\!]) = [\![b]\!]_{\bmod 2}$ where $b = 1$ iff $x < y$
Input: Shared values $[\![x]\!]$, $[\![y]\!]$ over $\mathbb{Z}_{2^\ell}$
Result: Shared bit $[\![b]\!]$

```
1: Compute $[\![x']\!]_{\bmod 2} = \mathsf{ExtractBit}([\![x]\!], \ell)$
2: Compute $[\![y']\!]_{\bmod 2} = \mathsf{ExtractBit}([\![y]\!], \ell)$
3: Compute $[\![d]\!] = [\![x]\!] - [\![y]\!]$
4: Compute $[\![d']\!]_{\bmod 2} = \mathsf{ExtractBit}([\![d]\!], \ell)$
5: Compute $[\![b_1]\!] = ([\![x']\!] \oplus [\![y']\!]) \wedge [\![y']\!]$
6: Compute $[\![b_2]\!] = \neg([\![x']\!] \oplus [\![y']\!]) \wedge [\![d']\!]$
7: return $[\![b]\!] = [\![b_1]\!] \oplus [\![b_2]\!]$
```

Theorem 26. *The unsigned less-than comparison protocol in Protocol 20 is correct.*

Proof. We first extract the most significant bits of the arguments x, y and their difference $x - y$. When calculating b_1, we consider the case, where the most significant bits of x and y differ ($x' \oplus y' = 1$). Then, if y has the non-zero bit, we have that $x < y$. Alternatively, if the most significant bits of x and y are equal, we check the most significant bit of the difference $x-y$. If $x < y$ then $x-y$ underflows and the most significant bit must be non-zero. This condition is computed as b_2. Finally we XOR b_1 and b_2 together to get the result, as only one of these two conditions can hold at the same time. $\square$

Note that the three invocations of $\mathsf{ExtractBit}$ can be called in parallel to decrease round-complexity. The two conjunctions for computing b_1 and b_2 can be run in parallel as well. Therefore, the total round-complexity is $\log_2 \ell + 2$, as we perform an $\mathsf{Overflow}$ and a bitwise conjunction. In total, we consume $3\ell(\log_2 \ell + 1) + 2\ell$ bit conjunction triples.

For signed comparison in Protocol 21, we first flip the most significant bit of the arguments, which represents the sign bit in two's complement representation, and then perform an unsigned comparison.

Theorem 27. *The signed less-than comparison protocol in Protocol 21 is correct.*

Proof. By flipping the sign bits, we effectively add $2^{\ell-1}$ to the value of both arguments, if we interpret them later as unsigned values instead. This means the new values are guaranteed to be positive, but retain their respective ordering. Thus, an unsigned comparison provides the desired result. $\square$

For bitwise shared integers, we can construct a more efficient protocol directly, without using the $\mathsf{Overflow}$ protocol. The protocol is presented as Protocol 22.

Protocol 21 Less-than comparison protocol of ℓ-bit signed additive types

Functionality: $\mathcal{F}([\![x]\!],[\![y]\!]) = [\![b]\!]_{\bmod 2}$ where $b = 1$ iff $x < y$
Input: Shared values $[\![x]\!]$, $[\![y]\!]$ over $\mathbb{Z}_{2^\ell}$ (interpreted in two's complement)
Result: Shared bit $[\![b]\!]$

```
1: $\mathcal{CP}_1$ computes $[\![x]\!]_1[\ell] = [\![x]\!]_1[\ell] \oplus 1$
2: $\mathcal{CP}_1$ computes $[\![y]\!]_1[\ell] = [\![y]\!]_1[\ell] \oplus 1$
3: Compute $[\![b]\!] = [\![x]\!] < [\![y]\!]$ using unsigned comparison protocol Protocol 20
4: return $[\![b]\!]$
```

Protocol 22 Less-than comparison protocol of ℓ-bit bitwise types

Functionality: $\mathcal{F}([\![x]\!],[\![y]\!]) = [\![b]\!]_{\bmod 2}$ where $b = 1$ iff $x < y$
Input: Shared values $[\![x]\!]$, $[\![y]\!]$ over $\mathbb{Z}_2^\ell$
Result: Shared bit $[\![b]\!]$

```
1: Compute $[\![d]\!] = [\![x]\!] \oplus [\![y]\!]$
2: Compute $[\![m]\!] = \mathsf{MSNZB}([\![d]\!])$
3: Compute $[\![b']\!] = [\![m]\!] \wedge [\![y]\!]$
4: Compute $[\![b]\!]_{\bmod 2} = \bigoplus_{i=1}^{\ell} [\![b']\!][i]$
5: return $[\![b]\!]$
```

Theorem 28. *The less-than comparison protocol for bitwise types in Protocol 22 is correct.*

Proof. We first extract the most significant bit of $x \oplus y$ as m. Clearly, m contains a non-zero bit exactly in the first most significant position where x and y differ. If we then perform a conjunction of m and y, we get the same non-zero bit in the result exactly if y had a non-zero bit at that location. If this is the case, then y had a non-zero bit in a higher position than x, which means that $x < y$. □

The bitwise comparison protocol has $\log_2 \ell + 1$ rounds and consumes $\ell(\log_2 \ell + 1)$ bit conjunction triples.

5.6 Conversion between data types

5.6.1 Conversion between additive and bitwise representation

For converting from the additive representation to bitwise, a convenient solution is to use a bitwise adder circuit to add the additive shares together into a bitwise representation. For the bitwise addition in Protocol 23, we employ a Kogge-Stone parallel prefix adder [KS73], following the current implementation of Sharemind's three-party protection domain. The advantage compared to a naive ripple carry adder is that we can compute the carry bits in a logarithmic number of rounds.

Among different parallel prefix adder constructions, the Kogge-Stone adder has minimal depth. We do not go into details of binary adders here for brevity, but refer the reader to a concise overview in [ARI].

Protocol 23 Addition protocol for ℓ-bit bitwise types

Functionality: $\mathcal{F}(\llbracket x \rrbracket, \llbracket y \rrbracket) = \llbracket z \rrbracket$, where $z = x + y$
Input: Bitwise shared values $\llbracket x \rrbracket$, $\llbracket y \rrbracket$ over $\mathbb{Z}_2^\ell$
Result: Bitwise shared value $\llbracket z \rrbracket$

```
1: Compute $\llbracket p \rrbracket = \llbracket x \rrbracket \oplus \llbracket y \rrbracket$
2: Compute $\llbracket c \rrbracket = \llbracket x \rrbracket \wedge \llbracket y \rrbracket$
3: Fix $\llbracket p' \rrbracket = \llbracket p \rrbracket$
4: for $i \in \{1, \ldots, \log_2 \ell\}$ do
5:     Compute $\llbracket c' \rrbracket = \llbracket c \rrbracket \lll 2^{i-1}$
6:     Compute $\llbracket c \rrbracket = \llbracket c \rrbracket \oplus (\llbracket c' \rrbracket \wedge \llbracket p' \rrbracket)$
7:     Compute $\llbracket p' \rrbracket = \llbracket p' \rrbracket \wedge \Big( (\llbracket p' \rrbracket \lll 2^{i-1}) \vee \Big( 2^{2^{i-1}} - 1 \Big) \Big)$
8: end for
9: Compute $\llbracket c \rrbracket = \llbracket c \rrbracket \lll 1$
10: return $\llbracket z \rrbracket = \llbracket p \rrbracket \oplus \llbracket c \rrbracket$
```

Note that disjunction with a public constant is a local operation and that the two conjunctions performed in the for-cycle can be performed in parallel. Thus the bitwise addition protocol has $\log_2 \ell + 1$ rounds and uses a total of $\ell + 2\ell \log_2 \ell$ bit conjunction triples.

Given a bitwise addition protocol, we can simply add shares $\llbracket x \rrbracket_1$ and $\llbracket x \rrbracket_2$ together, by considering them as separate bitwise shared values, where the other party has a zero share. The protocol is presented as Protocol 24.

Protocol 24 Protocol for converting ℓ-bit additive type to bitwise representation

Functionality: $\mathcal{F}(\llbracket x \rrbracket_{\bmod 2^\ell}) = \llbracket y \rrbracket_{\oplus 2\ell}$, where $x = y$
Input: Additively shared value $\llbracket x \rrbracket$ over $\mathbb{Z}_{2^\ell}$
Result: Bitwise shared value $\llbracket y \rrbracket$ over $\mathbb{Z}_2^\ell$

```
1: Initialize bitwise shared values $\llbracket x' \rrbracket_{\oplus 2^\ell}$ and $\llbracket x'' \rrbracket_{\oplus 2^\ell}$
2: $\mathcal{CP}_1$ fixes $\llbracket x' \rrbracket_1 = \llbracket x \rrbracket_1$ and $\llbracket x'' \rrbracket_1 = 0$
3: $\mathcal{CP}_2$ fixes $\llbracket x' \rrbracket_2 = 0$ and $\llbracket x'' \rrbracket_2 = \llbracket x \rrbracket_2$
4: Compute $\llbracket y \rrbracket = \llbracket x' \rrbracket + \llbracket x'' \rrbracket$ using the bitwise addition protocol Protocol 23
5: return $\llbracket y \rrbracket$
```

Theorem 29. *The additive to bitwise conversion protocol in Protocol 24 is correct.*

Proof. Correctness follows directly from the bitwise addition protocol, since

$$y = [\![x']\!]_1 + [\![x'']\!]_1 + [\![x']\!]_2 + [\![x'']\!]_2 = [\![x]\!]_1 + [\![x]\!]_2$$

□

To convert bitwise shared values back to the additive representation, we follow a simple approach. We first convert each bit in the bitwise representation to an additive integer shared over $\mathbb{Z}_{2^\ell}$. Then we can use only local operations to add the individual bits together to get the total value of the integer in additive shares.

We therefore require a protocol to convert values shared over $\mathbb{Z}_2$ to values shared over a larger domain $\mathbb{Z}_{2^\ell}$. We adapt the protocol of [BNTW12] with a simpler composition, invoking the multiplication protocol directly.

If we naively convert the individual bit shares to a larger domain, we will get the wrong result in the case where both shares are 1, which gives the value 0 mod 2. Therefore, we can check for this case separately, by multiplying the individual shares as separate values. The protocol of [BNTW12] actually performs a similar operation, by cleverly inlining a reshare between two parties into the protocol. Our protocol is given in Protocol 25.

Protocol 25 Protocol for converting secret-shared bits to ℓ-bit additive types

Functionality: $\mathcal{F}([\![b]\!]_{\bmod 2}) = [\![x]\!]_{\bmod 2^\ell}$, where $b = x$
Input: Secret-shared bit $[\![b]\!]$ over $\mathbb{Z}_2$
Result: Additively shared value $[\![x]\!]$ over $\mathbb{Z}_{2^\ell}$

```
1: Initialize additively shared values $[\![x']\!]_{\bmod 2^\ell}$ and $[\![x'']\!]_{\bmod 2^\ell}$
2: $\mathcal{CP}_1$ fixes $[\![x']\!]_1 = [\![b]\!]_1$ and $[\![x'']\!]_1 = 0$
3: $\mathcal{CP}_2$ fixes $[\![x']\!]_2 = 0$ and $[\![x'']\!]_2 = [\![b]\!]_2$
4: Compute $[\![x]\!] = [\![x']\!] + [\![x'']\!] - 2 \cdot [\![x']\!] \cdot [\![x'']\!]$
5: return $[\![x]\!]$
```

Theorem 30. *The protocol for converting secret-shared bits to additive types in Protocol 25 is correct.*

Proof. Correctness can be shown simply with

$$x = [\![b]\!]_1 + [\![b]\!]_2 - 2 \cdot [\![b]\!]_1 \cdot [\![b]\!]_2 = ([\![b]\!]_1 + [\![b]\!]_2) \bmod 2$$

□

Protocol 26 Protocol for converting ℓ-bit bitwise type to additive representation

Functionality: $\mathcal{F}([\![x]\!]_{\oplus 2^\ell}) = [\![y]\!]_{\bmod 2^\ell}$, where $x = y$
Input: Bitwise shared value $[\![x]\!]$ over $\mathbb{Z}_2^\ell$
Result: Additively shared value $[\![y]\!]$ over $\mathbb{Z}_{2^\ell}$

```
1: for $i \in \{1, \dots, \ell\}$ do
2:     Compute $[\![b]\!]_{\bmod 2} = [\![x]\!][i]$
3:     Compute $[\![k_i]\!]_{\bmod 2^\ell} = [\![b]\!]$ using Protocol 25
4: end for
5: Compute $[\![y]\!] = \sum_{i=1}^{\ell} 2^{i-1} \cdot [\![k_i]\!]$
6: return $[\![y]\!]$
```

The boolean to additive conversion requires only a single round and uses one ℓ-bit multiplication triple.

We now use the boolean to additive conversion to construct the full bitwise to additive conversion protocol in Protocol 26.

Theorem 31. *The bitwise to additive conversion protocol in Protocol 26 is correct.*

Proof. Correctness follows trivially from the bitwise representation of integers, since we have $y = \sum_{i=1}^{\ell} 2^{i-1} \cdot x[i] = x$. □

We note that the bit conversions can all be done in parallel, which means this protocol has only a single round. However, it consumes exactly ℓ multiplication triples of bit-length ℓ.

5.6.2 Casting up and down

In some applications, we may require to cast values to shorter or wider bit-lengths. For example, if we know that an integer value belongs to a short range, we can perform some computations on smaller bit-length values, which is more efficient.

For bitwise types, casting is a trivial local operation. To cast to a higher bit-length value, we simply add zeroes to both shares as the most significant bits. To cast down, we instead throw away the most significant bits.

For additive types, casting down works the same way since $([\![x]\!]_1 \bmod 2^k) + ([\![x]\!]_2 \bmod 2^k) = ([\![x]\!]_1 + [\![x]\!]_2) \bmod 2^k$. However, when casting additive values up, we need to take care of the possible overflow that the shares may have with the smaller modulus. We also have to separately consider the unsigned and signed cases. The two protocols presented here are based on Sharemind's current implementation of the respective three-party protocols [KLR16].

Protocol 27 Protocol for casting unsigned ℓ-bit additive type to k-bit additive type, $k > \ell$

Functionality: $\mathcal{F}([\![x]\!]_{\bmod 2^\ell}) = [\![y]\!]_{\bmod 2^k}$, where $x = y$ and $k > \ell$
Input: Additively shared value $[\![x]\!]$ over $\mathbb{Z}_{2^\ell}$
Result: Additively shared value $[\![y]\!]$ over $\mathbb{Z}_{2^k}$ where $k > \ell$

```
1: Compute $[\![b']\!]_{\bmod 2} = \mathsf{Overflow}([\![x]\!], \ell)$
2: Compute $[\![b]\!]_{\bmod 2^k} = [\![b']\!]$ using boolean to additive conversion Protocol 25
3: Initialize additively shared value $[\![y]\!]_{\bmod 2^k}$
4: Both $\mathcal{CP}_i$ compute $[\![y]\!]_i = \big([\![x]\!]_i - 2^\ell \cdot [\![b]\!]_i\big) \bmod 2^k$
5: return $[\![y]\!]$
```

Theorem 32. *The protocol for casting up unsigned additive types in Protocol 27 is correct.*

Proof. The idea of the protocol is simple, we check whether $[\![x]\!]_1 + [\![x]\!]_2$ overflows the modulus 2^ℓ. If it does not, then we simply assign $[\![y]\!]_i = [\![x]\!]_i$. If it does, we additionally subtract 2^ℓ from the value under the higher modulus, to simulate the overflow. $\square$

The protocol has $\log_2 \ell + 2$ rounds as we do an Overflow and one multiplication for converting the shared bit to an additive type. In total, we consume $\ell \cdot (\log_2 \ell + 1)$ bit conjunction triples and a single k-bit multiplication triple.

For signed integers, we additionally have to take care to shift the sign bit to the correct position when casting up. Therefore, the main approach is similar to the unsigned case, but we have to check for existence of the sign bit and also overflow of the additive representation separately. The protocol is presented as Protocol 28.

Protocol 28 Protocol for casting signed ℓ-bit additive type to k-bit additive type, $k > \ell$

Functionality: $\mathcal{F}([\![x]\!]_{\bmod 2^\ell}) = [\![y]\!]_{\bmod 2^k}$, where $x = y$ and $k > \ell$
Input: Additively shared value $[\![x]\!]$ over $\mathbb{Z}_{2^\ell}$ (interpreted in two's complement)
Result: Additively shared value $[\![y]\!]$ over $\mathbb{Z}_{2^k}$ where $k > \ell$

```
1: Compute $[\![s']\!]_{\bmod 2} = \mathsf{ExtractBit}([\![x]\!], \ell)$
2: Compute $[\![b']\!]_{\bmod 2} = \mathsf{Overflow}([\![x]\!], \ell)$
3: Compute $[\![s]\!]_{\bmod 2^k} = [\![s']\!]$ using Protocol 25
4: Compute $[\![b]\!]_{\bmod 2^k} = [\![b']\!]$ using Protocol 25
5: Initialize additively shared value $[\![y]\!]_{\bmod 2^k}$
6: Both $\mathcal{CP}_i$ compute $[\![y]\!]_i = \big([\![x]\!]_i - 2^\ell \cdot [\![b]\!]_i + (2^k - 2^\ell) \cdot [\![s]\!]_i\big) \bmod 2^k$
7: return $[\![y]\!]$
```

Theorem 33. *The protocol for casting up signed additive types in Protocol 28 is correct.*

Proof. The protocol first extracts the sign bit as s'. Then, we compute as b' whether the two additive shares overflow. If both are false, the shares of the arguments are also valid for the result. However, if the sign bit is set, we should explicitly unset the sign bit at the current position ℓ and add it to the position k. This is exactly achieved by adding $(2^k - 2^\ell) \cdot [\![s]\!]$ to the result.

If the additive shares of the argument overflow, we perform the same correction as in the unsigned case, and subtract 2^ℓ from the result, to simulate the overflow. Due to properties of two's complement notation, the correct signed value is preserved. □

The protocol has $\log_2 \ell + 2$ rounds and uses $2\ell \cdot (\log_2 \ell + 1)$ bit conjunction triples and 2 k-bit multiplication triples. However, notice that the overflow computation for the sign bit can actually be optimized, as we can perform a conjunction on both shares of the sign bit and the overflow bit for the previous bits.

5.7 Summary of protocols

We present a summary of all protocols described in this section in Table 11. We exclude operations that can be done using only local operations. We present the round-complexity and also number of multiplication and bit conjunction triples used, assuming a single invocation on ℓ-bit data types. All communication performed in the protocols happens only in multiplication and bitwise conjunction. This means that total communication can be directly calculated from the triple usage.

Compared to the three-party protection domain, we have not yet implemented integer protocols for division, division with public constant and bit-shifts left and right for private shift values. Note that although many of our protocols use only conjunction triples, the division protocol for example, relies heavily on multiplication [BNTW12]. From more high-level primitives, we are missing an oblivious shuffle protocol. In the two-party setting, we cannot use the communication-efficient protocol from [LWZ11], which relies on secret sharing one party's secret among the other parties, which requires at least three parties in total. However, we can base shuffling on matrix multiplication or oblivious sorting [LWZ11].

From Table 11, we can compare the efficiency of performing computations on additive or bitwise types. We first point out that with the exception of the bitwise addition, non of the protocols are redundant. For a single operation, it is always less efficient to switch from one representation to the other, and perform the operation there, than it is to perform the operation on the original shares.

For example, the additive less-than comparison protocol is more efficient than converting both arguments to bitwise representation and performing the bitwise comparison[16]. However, when many operations need to be performed, that are easier to do in the other representation, this conversion can still pay off in a larger algorithm. Note that it is also possible to keep both representations of the same value for performing different operations.

Also, the ExtractBit protocol is useful only if a single certain bit needs to be extracted from an additive value. If already two bits are required, converting the whole integer to bitwise representation is more efficient. Interestingly, the only operation that is equally efficient in both representations is the equality protocol.

5.8 Implementing floating point operations and future directions

We have a number of options for adding floating point operations to `shared2p`. First, we can get IEEE 754 compliant floating point arithmetic easily, following the methods of [PS15], which is based on Yao's garbled circuits protocol evaluating Boolean circuits that implement floating point operations. In this approach, we use 32-bit and 64-bit bitwise shared representations for floating point numbers.

For Yao's protocol we also need a two-party oblivious transfer protocol. Since we are using the free-XOR optimization for garbling circuits [KS08] (allowing to evaluate XOR gates without communication), we can employ the correlated OT extension protocol as outlined in [ALSZ13]. However, in that case, it is not clear that we can rely only on input privacy, and proving security for the whole Yao's protocol with correlated OT might require the random oracle model. However, we can always use the protocol of [ALSZ13] for standard $\binom{2}{1}$-OT extension, which can be shown secure by using the correlation robustness property. We can also improve on the implementation by using the recent more efficient garbling scheme construction from [ZRE15], which is also compatible with free-XOR.

Alternatively, we can approach floating-point operations with the additive approach of [KW14], which has been improved and optimized in [LR15, KLR16]. The difficulty here is that for some protocols, we require multiplication of large bit-length integers for efficiency (up to a few-hundred bits). This simply means we have to precompute multiplication triples for many different bit-lengths.

A more general improvement for future work is to add support of the precomputation process into the Sharemind protocol language, which is used to generate the currently most efficient protocols for Sharemind's three-party protection do-

[16]For additive comparison, we use $3\ell \log_2 \ell + 5\ell$ bit triples. Converting both arguments and performing bitwise comparison uses $5\ell \log_2 \ell + 3\ell$. Also, round-complexity is higher by $\log_2 \ell$ rounds.

Table 11: Summary of online protocols in the `shared2p` protection domain.

Data type	**Protocol**	**Rounds**	**Mult. triples**	**Bit conj. triples**	**Ref.**
`uint/int`	$[\![x]\!] \cdot [\![y]\!]$	1	1 ℓ-bit triple	-	Prot. 14
`uint/int`	$[\![x]\!] \overset{?}{<} [\![y]\!]$	$\log_2 \ell + 2$	-	$3\ell \cdot (\log_2 \ell + 1) + 2\ell$	Prot. 20, Prot. 21
`uint/int`	$[\![x]\!] \overset{?}{=} [\![y]\!]$	$\log_2 \ell$	-	$\ell - 1$	Prot. 16
`uint/int`	Extracting $[\![x]\!][k]$	$\log_2 \ell + 1$	-	$\ell \cdot (\log_2 \ell + 1)$	Prot. 19
`uint/int`	convert $[\![x]\!]$ to bitwise	$\log_2 \ell + 1$	-	$\ell \cdot (2\log_2 \ell + 1)$	Prot. 24
`uint`	cast $[\![x]\!]_{\mathrm{mod}\ 2^\ell}$ to $[\![x]\!]_{\mathrm{mod}\ 2^k}$, $k > \ell$	$\log_2 \ell + 2$	1 k-bit triple	$\ell \cdot (\log_2 \ell + 1)$	Prot. 27
`int`	cast $[\![x]\!]_{\mathrm{mod}\ 2^\ell}$ to $[\![x]\!]_{\mathrm{mod}\ 2^k}$, $k > \ell$	$\log_2 \ell + 2$	2 k-bit triples	$2\ell \cdot (\log_2 \ell + 1)$	Prot. 28
`xor_uint`	$[\![x]\!] + [\![y]\!]$	$\log_2 \ell + 1$	-	$\ell \cdot (2\log_2 \ell + 1)$	Prot. 23
`xor_uint`	$[\![x]\!] \wedge [\![y]\!]$	1	-	ℓ	Analogous to Prot. 14
`xor_uint`	$[\![x]\!] \vee [\![y]\!]$	1	-	ℓ	Prot. 15
`xor_uint`	$\mathsf{MSNZB}([\![x]\!])$	$\log_2 \ell$	-	$\ell \cdot \log_2 \ell$	Prot. 17
`xor_uint`	$[\![x]\!] \overset{?}{<} [\![y]\!]$	$\log_2 \ell + 1$	-	$\ell \cdot (\log_2 \ell + 1)$	Prot. 22
`xor_uint`	$[\![x]\!] \overset{?}{=} [\![y]\!]$	$\log_2 \ell$	-	$\ell - 1$	Analogous to Prot. 16
`bool`	convert $[\![b]\!]_{\mathrm{mod}\ 2}$ to additive $[\![x]\!]_{\mathrm{mod}\ 2^\ell}$	1	1 ℓ-bit triple	-	Prot. 25
`xor_uint`	convert $[\![x]\!]$ to additive	1	ℓ ℓ-bit triples	-	Prot. 26

main [LR15]. This requires small additions to the language and compiler, to use precomputed triples in the protocol description. With this approach, we gain from the automatic optimizations done by the protocol language compiler, and can improve the efficiency compared to the currently handwritten C++ protocols.

5.9 Implementation and benchmarks

5.9.1 Setting up the precomputation

When building applications on `shared2p`, an important technical implementation issue is how to ensure a sufficient amount of precomputation results are always available for the online protocols. In our current implementation, we run the precomputation as a separate thread that constantly performs the triple generation protocol, and saves the results to a buffer of predetermined size. If the buffer is full, the precomputation thread waits until a sufficient amount of triples are used from the buffer by the online protocols, that may be running in parallel, and then starts performing the triple computation protocol again. The current set-up is rather crude in the sense that even if the buffer size is large, the triples can run out quite quickly in a longer computation and then the precomputation running in parallel affects the performance of the online phase.

In our current implementation, when the precomputation is empty, an online protocol simply blocks and waits until more triples are precomputed. Another possibility would be to run and independent triple generation protocol in parallel, so that the online computation could continue. However, the complexity is in synchronizing the state of the PRG-s and concurrent accesses to them. We currently have mostly ignored these topics, but we stress that for a practical implementation, to support many parallel running computations, these issues have to be solved in a reasonable way.

5.9.2 Online performance benchmarks

We performed benchmarks of many of the protocol presented in the previous section. Specifically, we benchmark multiplication, comparison and equality protocols, conversions from additive to bitwise and vice versa, and also protocols for casting up unsigned and signed additive types. The benchmarks were performed on the same cluster of two machines as described in Section 4.6.1. We present these benchmarks only in the LAN setting, that is, we do not throttle traffic and use the 10 Gbit/s bandwidth network. We performed the benchmarks for all different bit-length data types. To measure the online performance, we manually turned off the precomputation and made the protocols use hardcoded triple values. We present the performance for the `shared2p` protocols in Table 12. We also performed benchmarks for the corresponding protocols in `shared3p`, presented in Table 13.

In all cases we report the average running times of 5 to 100 iterations, depending on the input data length. For smaller input sizes, we perform more iterations to reduce the variance of the result. Comparing the `shared2p` and `shared3p` online running times, we see that the results are in favor of `shared3p` in most cases. This is to be expected, since some of the `shared3p` protocols are written in the special protocol language, which is able to perform various automatic optimizations to the protocol. Concretely, additive less-than comparison, additive equality, additive to bitwise conversion and both up cast protocols are implemented with the protocol language. However, surprisingly, for additive less-than comparison, the `shared2p` protocol even outperforms the three-party one on 32 and 64-bit integers.

In most cases, our two-party protocols outperform `shared3p` ones on scalars and vectors of 10 elements. This can probably be explained by the inherent reduced latency in the two-party setting, where both parties only communicate with a single party. In the three-party setting, they have to exchange messages with both of the other parties. Since scalar operations are very fast, this can have a relevant effect on performance.

Overall, `shared2p` protocols are at most 2 times slower than the `shared3p` protocols, except for the case of signed up cast, for which the `shared2p` version is up to 3 times slower. We note that this is not a completely fair comparison, due to the fact that our protocols are hand-written in C++, and some of the `shared3p` protocols are heavily optimized by the protocol language compiler. However, the two-party multiplication and conjunction protocols, for which all the others rely on, use exactly twice as much communication between a pair of parties. However, the three-party protocols perform this amount of communication with both parties. In the future, it would be very interesting to see how much the two-party protocols can be optimized by the protocol language compiler.

We make a comparison also with the ABY framework [DSZ15]. The authors of [DDK$^+$15] report amortized performance for online operations with 32-bit integers on input size 10^4. Their test environment however uses a 1Gbit/s connection. For additive multiplication they report amortized 2 μs single operation time in the LAN setting. For the same input size, our multiplication has 0.065 ns as amortized time for a single operation, which is two orders of magnitude faster. For less-than comparison of bitwise 32-bit types, they report 4 μs amortized time, as opposed to our 0.47 ns, which is one order of magnitude difference. Arguably, the bandwidth differences have an effect on these times. However, we believe our much better multiplication performance is mostly due to the mature and optimized network layer of Sharemind, which is being constantly optimized and improved. Overall, we see that our two-party computation online performance is very competitive.

Table 12: Online performance of `shared2p` protocols for different data type bit lengths ℓ and input vector sizes. Performance is presented in `ops/ms` (amortized number of operations performed in one millisecond).

Protocol	ℓ	10^0	10^1	10^2	10^3	10^4	10^5	10^6
Additive $[\![x]\!] \cdot [\![y]\!]$	8	7.06	72.00	706.26	6145.53	25898.68	38704.93	40419.02
	16	6.70	68.15	618.58	5573.83	19303.53	22627.20	23822.34
	32	6.08	63.21	622.67	4291.85	15257.86	15096.66	15882.03
	64	4.19	42.51	418.43	3738.60	8960.09	8927.23	9668.30
Additive $[\![x]\!] \overset{?}{<} [\![y]\!]$	8	1.05	10.52	88.37	493.58	1203.93	1495.52	1502.61
	16	0.85	8.22	64.97	364.36	858.15	939.27	961.20
	32	0.74	7.09	51.85	270.99	538.26	573.19	609.67
	64	0.64	5.90	39.52	196.48	254.51	304.14	312.72
Bitwise $[\![x]\!] \overset{?}{<} [\![y]\!]$	8	3.93	38.61	342.08	2367.93	7341.12	8544.63	8803.54
	16	2.81	25.76	220.03	1501.98	4216.47	4086.69	4201.66
	32	2.18	20.65	172.78	973.93	2091.34	1858.27	2150.43
	64	1.61	16.27	118.42	611.58	850.02	940.42	1045.34
Additive $[\![x]\!] \overset{?}{=} [\![y]\!]$	8	3.31	32.76	303.32	2206.53	6729.97	8347.48	8974.48
	16	3.11	31.10	289.14	1944.13	5468.60	6448.09	7112.64
	32	2.32	22.57	206.23	1426.09	4065.16	4635.36	5207.05
	64	2.25	21.55	189.57	1227.70	3035.51	3256.93	3657.11
Convert additive to bitwise	8	1.83	17.62	156.41	1037.52	3036.40	3619.07	4001.66
	16	1.45	13.93	115.14	757.31	1993.35	2110.98	2163.87
	32	1.16	11.26	92.05	506.88	1098.07	1127.07	1199.94
	64	1.08	8.91	72.15	342.12	519.58	608.81	657.30
Convert bitwise to additive	8	4.55	36.62	362.48	1818.81	3103.21	3133.50	3203.11
	16	4.03	38.90	299.65	1037.87	1086.82	1107.83	1135.85
	32	3.67	41.32	208.30	314.43	331.54	375.22	362.11
	64	3.17	23.02	96.04	94.64	108.44	111.03	118.19
Additive unsigned cast to `uint64`	8	1.72	18.09	165.02	959.69	2425.60	2762.31	3059.22
	16	2.03	20.65	167.53	885.47	2028.00	2077.15	2144.85
	32	1.58	15.68	117.23	705.88	1245.53	1378.01	1478.49
Additive signed cast to `int64`	8	1.26	11.07	103.71	579.83	1183.23	1359.97	1496.21
	16	0.95	10.67	80.59	471.24	950.01	1011.55	1078.78
	32	0.90	8.03	65.79	340.09	629.83	702.69	733.06

Table 13: Online performance of `shared3p` protocols for different data type bit lengths ℓ and input vector sizes. Performance is presented in `ops/ms` (amortized number of operations performed in one millisecond).

Protocol	ℓ	10^0	10^1	10^2	10^3	10^4	10^5	10^6
Additive $[\![x]\!] \cdot [\![y]\!]$	8	7.56	79.44	736.43	6869.55	38740.17	69201.28	61820.97
	16	6.14	66.12	601.61	5647.17	26848.52	38274.00	34296.48
	32	4.51	50.19	432.66	3888.02	17132.09	20340.33	18602.36
	64	4.31	35.34	364.79	2831.98	10186.20	11673.96	10176.84
Additive $[\![x]\!] \overset{?}{<} [\![y]\!]$	8	0.80	8.86	76.33	502.52	1365.30	1617.49	1783.62
	16	1.05	11.53	87.30	488.03	969.07	1005.56	1126.29
	32	0.88	8.64	64.84	317.43	558.50	504.16	551.18
	64	0.79	7.63	56.44	203.58	281.52	246.60	264.53
Bitwise $[\![x]\!] \overset{?}{<} [\![y]\!]$	8	2.50	26.44	246.88	1901.50	8835.17	11330.52	11480.99
	16	2.04	20.11	187.49	1365.47	4996.25	5856.54	5747.02
	32	1.74	17.01	147.87	1042.56	2571.13	2922.94	2897.70
	64	1.44	13.25	116.92	746.06	1272.51	1468.58	1422.55
Additive $[\![x]\!] \overset{?}{=} [\![y]\!]$	8	2.26	24.24	219.83	1963.02	8883.83	12295.98	14329.46
	16	2.03	20.81	204.81	1594.03	6753.38	8871.63	10385.50
	32	1.84	18.24	178.44	1405.80	4806.26	5815.51	6164.15
	64	1.73	17.73	157.08	1165.42	3212.39	3365.28	3689.35
Convert additive to bitwise	8	1.59	16.41	165.56	1272.35	5768.41	8201.81	9487.87
	16	1.21	11.37	116.46	807.80	3342.63	4221.26	5003.84
	32	1.53	16.05	130.55	871.84	1795.77	1780.35	2071.21
	64	0.87	9.49	82.43	464.17	902.64	736.13	828.06
Convert bitwise to additive	8	2.18	28.84	290.95	1828.32	3466.42	4265.43	4424.16
	16	2.70	26.52	211.10	1098.96	1458.23	1584.76	1784.77
	32	2.61	16.83	208.11	459.38	550.80	575.05	548.51
	64	1.08	20.14	117.41	157.14	191.46	189.22	185.93
Additive unsigned cast to `uint64`	8	0.93	10.06	93.00	661.22	2840.29	3767.73	4879.54
	16	0.87	8.89	88.20	636.30	2288.17	2922.01	3389.51
	32	0.71	5.17	53.35	449.64	1455.71	1704.01	1831.34
Additive signed cast to `int64`	8	0.77	8.70	85.67	548.30	2789.72	3683.72	5084.32
	16	0.67	7.36	56.28	622.05	2099.64	2351.79	2694.11
	32	0.86	7.85	77.68	507.51	1442.02	1365.62	1657.24

6 Conclusion

The difference between theory and practice often rests on one major factor: efficiency.

Donald Beaver

During the writing of this thesis, the author found this quote by Donald Beaver from his seminal paper of 1991, as a source of inspiration and a compact formulation of the principal motivation behind this work. Although the quote is as old as the author, one can argue that despite numerous advances in both theoretical and practical methods, this viewpoint is relevant in the field of cryptography to this day.

Our main goal in this work was to implement an efficient two-party secure computation protocol suite on Sharemind, that is competitive in performance with the three-party alternative, in order to provide a viable alternative to build applications that are not suited to the three-party model. We presented the design of our proposed protocol suite, with both efficient constructions for the offline phase, based on oblivious transfer extensions and a suite of online arithmetic protocols for integer and Boolean operations, as well as conversions between different secret sharing representations.

The main focus of the thesis was on optimizing the precomputation phase, as it is much more computationally intensive and the online phase highly depends on it. We have presented our constructions for Beaver triple generation that employ novel techniques for reducing communication costs, compared to similar methods used in previous work. We achieve this on a high level by using a smaller number of 1-out-of-N oblivious transfer instead of performing many 1-out-of-2 oblivious transfers. We have described and implemented these optimized Beaver triple generation protocols and benchmarked their performance. Overall, our techniques show promise in increasing the performance of the precomputation phase. However, our current implementation can be improved in a number of ways, since the local computations proved to be a bottleneck. Especially, we could improve on implementing a parallel batching technique, which allows to naturally interleave local computations and network communication. By reducing the overhead of local computations, our techniques, which have reduced communication, would have a larger effect on overall performance.

In showing security of our precomputation protocols, an important advancement is that we do not require the random oracle model to show security, but rely on concrete assumptions on hash functions. We prove the main results that are needed to show security of the composition of our input private oblivious transfer extension protocols with the secure reshare protocol. However, a full formal proof of the final composition result is left as future work. For the 1-out-of-N oblivious transfer extension protocol, we generalized the formalization of correlation robustness that was needed to show security, as the original paper lacked a security proof.

We also implemented a representative amount of computation protocols for the online phase and showed that they achieve the goal set out in being competitive with Sharemind's three-party protocols. In some cases, the two-party protocols are even faster. Comparing our work to a similar two-party implementation in the passive security model, we showed an order of magnitude better performance. Thus, we can conclude that we have succeeded in an efficient implementation, that would allow two-party secure computations on Sharemind.

In future work we will add missing protocols for integer division and integer bit shifts and also add Yao's garbled circuits based floating-point arithmetic protocols. Additionally we will finalize the precomputation phase by implementing a suitable base OT protocol for the extensions.

References

[ABPP15] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. *IACR Cryptology ePrint Archive*, 2015:1039, 2015.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548. ACM, 2013.

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 673–701. Springer, 2015.

[ARI] ARITH research group, Aoki lab., Tohoku University. Hardware algorithms for arithmetic modules. `http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html`. Accessed on 16.05.2016.

[BCD$^+$09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security*, pages 257–266, 2008.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara,* *California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 478–492. IEEE Computer Society, 2013.

[BJoSV15] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the Estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *LNCS*, pages 227–234. Springer, 2015.

[BK12] Elaine B. Barker and John M. Kelsey. Sp 800-90a. recommendation for random number generation using deterministic random bit generators. Technical report, Gaithersburg, MD, United States, 2012.

[BKK$^+$16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *PoPETs*, 2016(3):117–135, 2016.

[BKL$^+$14] Dan Bogdanov, Liina Kamm, Sven Laur, Pille Pruulmann-Vengerfeldt, Riivo Talviste, and Jan Willemson. Privacy-preserving statistical data analysis on federated databases. In *Proceedings of the Annual Privacy Forum. APF'14*, volume 8450 of *LNCS*, pages 30–55. Springer, 2014.

[Bla79] George R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, pages 313–317, Monval, NJ, USA, 1979. AFIPS Press.

[BLLP14] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation primitives. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 184–198. IEEE, July 2014.

[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.

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

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

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73. ACM, 1993.

[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.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19. ACM, 1988.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro and Matthew Robshaw, editors, *Advances* *in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[CJS14] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 597–608, 2014.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2015.

[DDK$^+$15] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1504–1517. ACM, 2015.

[DDN$^+$15] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. *IACR Cryptology ePrint Archive*, 2015:1006, 2015.

[DKMR05] Anupam Datta, Ralf Küsters, John C. Mitchell, and Ajith Ramanathan. On the relationships between notions of simulation-based security. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 476–494. Springer, 2005.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances* *in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590. Springer, 2007.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[Ekl72] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21(7):801–803, July 1972.

[Gil99] Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 1999.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

[GOR11] Vipul Goyal, Adam O'Neill, and Vanishree Rao. Correlated-input secure hash functions. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 182–200. Springer, 2011.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[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.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 20–31. ACM, 1988.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2013.

[KLR16] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *to appear in: Financial Cryptography and Data Security - FC 2016 Workshops, BITCOIN, VOTING and WAHC, Barbados, February 26, 2016, Revised Selected Papers*, 2016.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, 2015.

[KS73] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, 22(8):786–793, 1973.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[KS16] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 699–728. Springer, 2016.

[KW14] Liina Kamm and Jan Willemson. Secure Floating-Point Arithmetic and Private Satellite Collision Analysis. *International Journal of Information Security*, pages 1–18, 2014.

[Lin08] Yehuda Lindell. Efficient fully-simulatable oblivious transfer. *Chicago J. Theor. Comput. Sci.*, 2008, 2008.

[LMS16] Helger Lipmaa, Payman Mohassel, and Seyed Saeed Sadeghian. Valiant's universal circuit: Improvements, implementation, and applications. *IACR Cryptology ePrint Archive*, 2016:17, 2016.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[LR15] Peeter Laud and Jaak Randmets. A domain-specific language for low-level secure multiparty computation protocols. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1492–1503. ACM, 2015.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 557–574. Springer, 2013.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In S. Rao Kosaraju, editor, *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 448–457. ACM/SIAM, 2001.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[PBS12] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. The design and implementation of a two-party protocol suite for Sharemind 3. Technical Report T-4-17, Cybernetica, `http://research.cyber.ee/`., 2012.

[PGFW14] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N. Wright. Systematizing secure computation for research and decision support. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*, pages 380–397. Springer, 2014.

[Plo60] Morris Plotkin. Binary codes with specified minimum distance. *IRE Trans. Information Theory*, 6(4):445–450, 1960.

[PS15] Pille Pullonen and Sander Siim. Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations. In *Financial Cryptography and Data Security - FC 2015 Workshops, BITCOIN, WAHC and Wearable 2015, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *LNCS*, pages 172–183. Springer, 2015.

[PW00] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati, editors, *CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1-4, 2000.*, pages 245–254. ACM, 2000.

[Rab81] Michael O. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University, 1981.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.

[Sec] Dyadic Security. Dyadic security white paper. Published online at `https://www.dyadicsec.com/wp-content/uploads/2015/06/dyadicwhitepaper.pdf`. Last accesed on 16.05.2016.

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

[Vah15] Meril Vaht. The analysis and design of a privacy-preserving survey system. Master's thesis, Institute of Computer Science, University of Tartu, 2015.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proc. of SFCS'82*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, 2015.

A Codewords that achieve the Plotkin bound

Here, we present the codewords that we used in our implementation of the [KK13] protocol. These codes achieve the Plotkin bound, which gives a minimum length κ for existence of n codewords whose pairwise minimum distance is d. We mean by $101\ldots$ that the same pattern 101 should be continued to fill the length of the codeword.

Table 14: Codewords used in our [KK13] protocol implementation.

d	n	κ	**Codewords**
128	2	128	000...
			111...
128	4	192	000...
			011...
			110...
			101...
128	8	224	0000000...
			0001111...
			0110011...
			0111100...
			1010101...
			1011010...
			1100110...
			1101001...
128	16	240	000000000000000...
			000000011111111...
			000111100001111...
			000111111110000...
			011001100110011...
			011001111001100...
			011110000111100...
			011110011000011...
			101010101100110...
			101010110011001...
			101101001101001...
			101101010010110...
			110011001010101...
			110011010101010...
			110100101011010...
			110100110100101...