UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Sander Siim

Secure Multi-party Computation Protocols from a High-Level Programming Language

Bachelor's Thesis (6 ECTS)

Supervisor: Dan Bogdanov, PhD
Supervisor: Sven Laur, PhD

Tartu 2014

Turvalise ühisarvutuse protokollid kõrgtaseme programmeerimiskeelest

Lühikokkuvõte: Turvalise ühisarvutuse abil on võimalik sooritada privaatsust säilitavaid arvutusi mitmelt osapoolelt kogutud andmetega. Tänapäeva digitaalses maailmas on andmete konfidentsiaalsuse tagamine üha raskemini teostatav. Turvalise ühisarvutuse meetodid nagu ühissalastus ja Yao sogastatud loogikaskeemid võimaldavad teostada privaatsust säilitavaid arvutusprotokolle, mis ei lekita konfidentsiaalseid sisendandmeid. Aditiivne ühissalastuse skeem on väga efektiivne algebraliste ringide tehete sooritamiseks fikseeritud bitilaiusega andmetüüpide peal. Samas on seda kasutades raske ehitada protokolle, mis nõuavad paindlikumaid bititaseme operatsioone. Yao sogastatud loogikaskeemide meetod töötab aga igasuguse bitilaiusega andmete peal ja võimaldab väärtustada mistahes Boole'i funktsioone. Neid kahte meetodit koos kasutades ehitame turvalise hübriidprotokolli, mis kujutab endast üldist meetodit privaatsust säilitavate arvutuste teostamiseks bitikaupa ühissalastatud andmete peal. Loogikaskeeme vajalikeks arvutusteks on lihtne saada kahe kaasaegse turvalise ühisarvutuse jaoks mõeldud kompilaatori abil, mis muundavad C programmi loogikaskeemiks — PCF ja CBMC-GC. Meie hübriidprotokolli prototüüp privaatsust säilitaval arvutusplatvormil Sharemind saavutab praktilisi jõudlustulemusi, mis on võrreldavad teiste kaasaegsete lahendustega. Lisaks kahe osapoolega arvutustele pakub meie prototüüp võimekust teostada mitmekesiseid arvutusi üldises turvalise ühisarvutuse arvutusmudelis. Hübriidprotokoll ja loogikaskeemide kompilaatorid võimaldavad koos kasutades lihtsalt ja efektiivselt luua üldkasutatavaid turvalise ühisarvutuse protokolle mistahes Boole'i funktsioonide väärtustamiseks.

Võtmesõnad: krüptograafia, krüptograafilised protokollid, andmete privaatsus, turvaline ühisarvutus, Yao sogastatud loogikaskeemid, ühissalastus, kompilaatorid, teostus, jõudlus

Secure Multi-party Computation Protocols from a High-Level Programming Language

Abstract: Secure multi-party computation (SMC) enables privacy-preserving computations on data originating from a number of parties. In today's digital world, data privacy is increasingly more difficult to provide. With SMC methods like secret sharing and Yao's garbled circuits, it is possible to build privacy-preserving computational protocols that do not leak confidential inputs to other parties. The additive secret sharing scheme is very efficient for algebraic ring operations on fixed bit-length data types. However, it is difficult to build protocols that require robust bit-level manipulation. Yao's garbled circuits approach, in contrast, works on arbitrary bit-length data and allows the evaluation of any Boolean function. Combining the two methods, we build a secure hybrid protocol, which provides a general method for building arbitrary secure computations on bitwise secret-shared data. We are able to generate circuits for the protocol easily by using two state-of-the-art C to circuit compilers designed for SMC applications — PCF and CBMC-GC. Our hybrid protocol prototype on the Sharemind privacy-preserving computational platform achieves practical performance comparable to other recent work. In addition to two-party computations, our prototype provides the ability to perform a set of diverse computations in a generic SMC computational model. The hybrid protocol together with the circuit compilers provides a simple and efficient toolchain to build general-purpose SMC protocols for evaluating any Boolean function.

Keywords: cryptography, cryptographic protocols, data privacy, secure multi-party computation, Yao garbled circuits, secret sharing, compilers, implementation, performance

Contents

1 Introduction

Secure multi-party computation (SMC) is a subfield of cryptography that studies methods for enabling multiple parties to compute functions on their joint inputs while preserving the privacy of those inputs. This can be achieved by building computational protocols that work on some sort of encrypted form of data, rather than computing on data directly.

The possible applications of such technology are wide, since privacy of data in the digital world is an everyday issue. People do not want their sensitive personal information being revealed and companies would like to protect their financial data. However, in today's world, we are often forced to make compromises between privacy and comfort. By using cryptographic privacy-preserving methods to manipulate with data, we can be more confident, that our personal data does not fall into unfriendly hands, while still being able to make use of the vast possibilities that the modern digital age provides.

The first theoretical solutions for SMC first appeared in the 1980-s, but the actual implementations have been far too inefficient for practical use, until recently. In the past 10 years, many platforms and frameworks that provide general-purpose SMC with viable performance have emerged [2, 19, 22, 33, 5]. Also, the first real-world practical applications which use SMC to provide data privacy have appeared in the past few years [10, 9]. It is clear that this new technology is developing fast and will reach a mature stage sometime soon in the future, as already a large number of real-world applications would benefit from using SMC technology.

Thus, it is important to constantly improve the state-of-the-art of SMC solutions, so that frameworks and applications that provide data privacy could be developed more easily and efficiently in the future. Not only are the provided theoretical security guarantees important, but also the efficiency and effort of building, maintaining and using these new solutions.

In this thesis, we focus on combining different existing theoretical principles and developed tools to easily build useful secure computation protocols with practical performance. The topic of this thesis was largely motivated by Oleg Šelajev's MSc thesis [43], where a general secure two-party SMC protocol was introduced. In this thesis, we build on the ideas of [43] to improve both the performance and applicability of the proposed protocol. Our presented hybrid protocol leverages many well-known optimizations from the literature and uses a more recent and refined garbling scheme technique, resulting in greatly improved performance of our implementation. Also, our protocol is designed to be used in a more generic SMC computational model and provides potentially arbitrary secure computations by using circuits generated with state-of-the-art circuit compilers. All together, our solution presents an efficient way to build new general-purpose SMC protocols in a semi-automated manner, requiring much less effort than it usually takes to

design such protocols.

Parts of this thesis have also been previously published in a recent research report [41].

Contributions of the author. The author of the thesis did research on current literature in the subject of Yao's garbled circuits protocols to find efficient state-of-the-art garbling methods and optimizations. Based on the findings, the author then formulated the hybrid protocol described in this thesis and composed a proof sketch of its security in the passive model. The author also implemented a complete prototype of the described protocol in the Sharemind `additive3pp` protection domain.

The author studied the state-of-the-art of automatic Boolean circuit generation and acquired, evaluated and benchmarked two circuit compilers from C programs — PCF and CBMC-GC. The author performed an analysis concerning the performance, functional capability and field of application of both compilers. Also, integration with the PCF circuit compiler was made to the hybrid protocol prototype by the author.

Finally, the author performed performance tests on the hybrid protocol prototype using different circuits and compared the results with other recent work in the field.

Thesis outline. In Section 2, we introduce the setting of secure two-party computation and present the widely used Yao's garbled circuits method, which is the first general solution for performing secure computations with multiple parties. We also discuss a more generic SMC computational model with more than two parties, and describe two other SMC methods that prove useful in this setting.

In Section 3, we compare two state-of-the-art Boolean circuit compilers - PCF and CBMC-GC - which are very convenient tools for building protocols using the garbled circuits method.

Then, combining two previously discussed SMC methods, we present an efficient general-purpose SMC protocol inspired by the work of [43] in Section 4. We also provide a proof sketch of the protocol's security in the passive model.

In Section 5 we describe our prototype implementation of the presented protocol built on the Sharemind platform, and in Section 6, analyze its performance and compare it to other similar results in the field.

2 Yao's garbled circuits

Yao's garbled circuits technique is a widely used and provably secure solution to the two-party secure computation problem [32]. It was first introduced by Andrew C.C. Yao in his seminal paper in 1982 [44]. Yao's protocol allows two parties to compute a function on their joint inputs without revealing one's inputs to the other.

In Yao's protocol, the mathematical model of *Boolean circuits* is used to represent the computed functions. A Boolean circuit is a set of gates and wires. Wires connect gates and transfer bit values between them. Each gate has a number of input wires and output wires and performs an elementary Boolean logic operation. If a gate has u input wires and v output wires, then the gate calculates a function $g : \{0,1\}^u \to \{0,1\}^v$. In practice, mostly 2-to-1 and 1-to-1 gates are used. The gate calculates the function by receiving the input bits from its input wires and sending the output to its output wires.

Boolean circuits can be used to describe any function $f : \{0,1\}^n \to \{0,1\}^m$, but a single circuit can calculate only one specific function. Formally, a circuit with its gates and connecting wires forms a directed acyclic graph. A circuit is given input through a number of external input wires, and the overall result is sent to the circuit's external output wires. Let C_f be a circuit which calculates the function $f : \{0,1\}^n \to \{0,1\}^m$. To calculate $y = f(x)$, we feed the value x encoded as a bitstring with length n to the circuit through its external input wires. Then we traverse all of the circuit's gates in a topological order while evaluating each gate with inputs from previous gates or the external wires. The final result will be sent to the circuit's external output wires, where it can be read from.

An example of a Boolean circuit is illustrated in Figure 1, which calculates the greater-than-or-equal function on one-bit values.

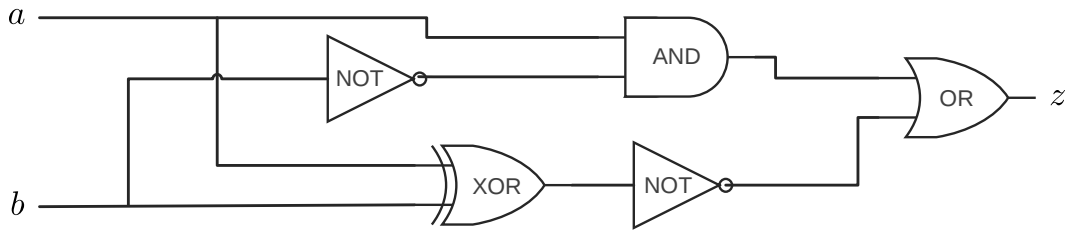

Figure 1: An example of a Boolean circuit. Calculates $z = a \geq b$

Using Yao's protocol, two parties can compute virtually any function in a privacy-preserving manner, provided that the Boolean circuit representation of the required function is known. Building efficient implementations of Yao's protocol has been a significant challenge in the past, but Yao's protocol has been much

refined since his 1982 paper and many succesful implementations have emerged in the past years [34, 22, 24, 29].

2.1 General technique

We now describe the standard modern view of Yao's protocol and how it succeeds in performing secure two-party computation [32].

The idea of Yao's protocol is to evaluate a Boolean circuit without actually seeing the bit values that run through wires during evaluation. In the protocol, the two parties assume different roles. One party will be the *garbler* and the other the *evaluator*. Both of these parties can provide input to the calculation.

In a nutshell, the garbler's task is to encrypt the chosen circuit so that it could be evaluated obliviously by the evaluator. By obliviously, we mean that the evaluator should not learn the other party's input to the circuit, nor any intermediate gate evaluation results. This is achieved in the protocol by three main techniques:

- The rows of gates' truth tables are randomly shuffled
- Instead of bit values, encryption scheme keys are transferred through wires and used to encrypt the shuffled truth tables
- Oblivious transfer is used to securely send correct input keys for evaluation to the evaluator

The setting for Yao's two-party protocol is the following. Parties $\mathcal{A}$ and $\mathcal{B}$ want to compute some function f on their respective secret inputs a and b. Both parties want to learn the result $f(a, b)$, but neither wants to disclose his input to the other party. Let us assume both parties have access to a Boolean circuit C_f which calculates the function f. Party $\mathcal{A}$ will take the role of the *garbler* and party $\mathcal{B}$ the *evaluator*.

The essence of Yao's protocol is that the garbler uses symmetric encryption or pseudo-random functions to encrypt the truth table values of the circuit's gates. This process is called *garbling*. For each wire, the garbler generates two tokens X^0, X^1 which represent the bit values of the wire — one token for bit value 0 and one for bit value 1. However, the token itself is random and does not reveal the bit that it maps to. The circuit can then be evaluated only if the evaluator knows the secret tokens of the input wires that correspond to the given input bits.

Since garbling is the most important part in the protocol, we will illustrate the process with an example of garbling an AND-gate. Consider the AND-gate in Figure 2 with input wires a, b and output wire z.

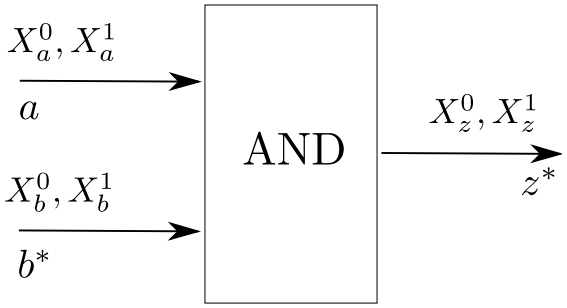

Figure 2: Garbling an AND-gate

a	b^*	$z^* = a \wedge b^*$
0	0	1
0	1	1
1	0	0
1	1	1

Table 1: Shuffled AND-gate truth table with wires b, z flipped

First, the garbler generates a token-pair for each wire, for example X_a^0 and X_a^1 correspond to the 0-bit and 1-bit respectively for wire a. Then, the garbler shuffles the truth table rows of the gate, so that the evaluator would not learn the result of the gate's computation during evaluation. If the rows were not shuffled, the index of the truth table row would immediately betray the gate's output during evaluation. To shuffle, the garbler "flips" the semantics of each wire with probability $\frac{1}{2}$. In our example, the semantics of wire b and z are flipped and we denote this as b^* and z^*. As we can see from Table 1, the 0 and 1 bit have flipped semantics for wires b and z. However, although the evaluator is not told which wires have been flipped, it can easily deduce this by looking at the truth table, since it knows that the garbled gate is an AND-gate.

Therefore, the next step for garbling the gate is to encrypt the truth table values. For this, the garbler uses an encryption scheme $\mathcal{E}$, which can encrypt a plaintext using two secret keys. In practice, a secure block cipher or hash function is used for this. The garbler will then encrypt the result column of the truth table using the generated wires' tokens as encryption keys. The resulting garbled truth table for our example is presented in Table 2. Here, we use $\mathcal{E}((A, B), X)$ to show that X is encrypted using keys A and B.

a	b^*	$z^* = a \wedge b^*$
X_a^0	X_b^0	$\mathcal{E}((X_a^0, X_b^0), X_z^1)$
X_a^0	X_b^1	$\mathcal{E}((X_a^0, X_b^1), X_z^1)$
X_a^1	X_b^0	$\mathcal{E}((X_a^1, X_b^0), X_z^0)$
X_a^1	X_b^1	$\mathcal{E}((X_a^1, X_b^1), X_z^1)$

Table 2: Garbled AND-gate truth table

Now, the evaluator is only able to decrypt the one truth table row for which

he possesses both corresponding tokens. As a result of the decryption, it receives a single token for the gate's output wire, but since the wire may have flipped semantics, the evaluator will not know which bit the token represents. Also, there is no way to find out whether the wire has been flipped, since the remaining truth table rows cannot be decrypted.

Now, for garbling a complete circuit, this process is continued with all gates. The output tokens of one gate are used as the input tokens of successive gates to encrypt the corresponding truth tables. We can now see, that if the evaluator possessed a set of tokens for the circuit's input wires - one token for each wire - he could evaluate the circuit obliviously by decrypting a single truth table row from each gate and using the results to decrypt truth tables of successive gates.

Let us remind that in our setting, we have parties $\mathcal{A}$ and $\mathcal{B}$ who each have their own inputs a and b respectively. Now, $\mathcal{A}$ has generated the tokens for the circuit's input wires and also garbled the circuit's truth tables. It is now necessary to send $\mathcal{B}$ the input tokens which correspond to the joint input of a and b. For the input tokens corresponding to a, the garbler $\mathcal{A}$ simply chooses the correct tokens according to his input and sends them to the evaluator. The input a will not be revealed to $\mathcal{B}$, since the semantics of the tokens is random.

Now, for $\mathcal{B}$ to receive the tokens corresponding to his own input, a well-known cryptographic primitive called *oblivious transfer* is used. Oblivious transfer guarantees that $\mathcal{B}$ will receive the correct tokens, while $\mathcal{A}$ remains oblivious as to which tokens it sent to $\mathcal{B}$, thereby preserving the privacy of $\mathcal{B}$'s input.

We have now described all the necessary components for parties $\mathcal{A}$ and $\mathcal{B}$ to execute Yao's protocol. The garbler $\mathcal{A}$ will garble the circuit's truth tables and send them to $\mathcal{B}$. Using oblivious transfer, all the correct input tokens are also sent to $\mathcal{B}$, who can then obliviously evaluate the circuit. An illustrative overview of the whole protocol is given in Figure 3.

As a result of this oblivious evaluation, the evaluator is left with a set of tokens for the circuit's external output wires. There are then two possibilities. The garbler can choose to flip the output wires similarly to all others, or leave the output wires unflipped. If the output wires are not flipped, than the output tokens will directly reveal the actual output of the computation to the evaluator. In the semi-honest model, the garbler is then also guaranteed to receive the output, as the evaluator simply sends it to him. However, if the output wires are also flipped, the evaluator will send the result of the evaluation back to the garbler, who can then deduce the real output, since it knows which wires were flipped. Again, if the garbler is honest, he will send the actual output also to the evaluator.

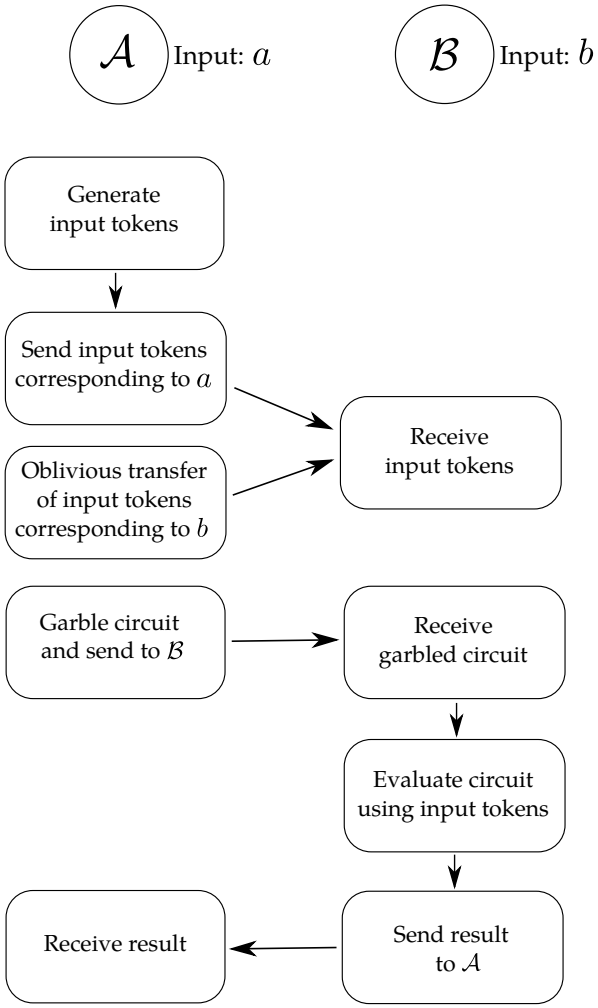

Figure 3: General construction of Yao garbled circuit protocols

2.2 Comparison with other secure computation techniques

We have seen that Yao's protocol enables us to perform secure two-party computation on a large number of functions. However, we might come across applications where a more general SMC setting is needed with an arbitrary number of parties. Yao's protocol is not specifically designed to work in these settings, although the protocol can be extended to be used in various other contexts [2]. Also, since Yao's protocol operates on a bit-level, the performed computations have a significant overhead compared to ordinary operations.

There are however other cryptographic methods with different advantages that can be used to perform secure computations. We will now discuss two other useful techniques - *secret sharing* and *homomorphic encryption*.

2.2.1 Secret sharing

Secret sharing is a well-known cryptographic method for distributing a secret amongst a group of parties [39, 4]. The secret is divided into *shares* and each party receives a share of the secret, which appears random to the receiving party. The goal for any secret sharing scheme is that the secret can only be reconstructed by combining a sufficiently large subset of the shares. For a *k-out-of-n secret sharing scheme*, the secrets are divided into n shares and knowing any $k-1$ shares does not reveal the original secret. Secret sharing schemes can be used in secure multi-party computation to build protocols that guarantee data privacy by performing computations on secret-shared data [3, 13].

We will describe the *additive secret sharing scheme* as one example of secret sharing. Let us assume that we want to secret-share a 32-bit integer value between n parties. Mathematically, k-bit integers are elements of the ring $\mathbb{Z}_{2^k}$. We use $x \xleftarrow{\$} \mathbb{Z}_{2^{32}}$ to denote that x is uniformly randomly sampled from the set $\mathbb{Z}_{2^{32}}$. We can construct random shares for some $x \in \mathbb{Z}_{2^{32}}$ in the following way [5]:

$$
\begin{aligned}
x_1 &\xleftarrow{\$} \mathbb{Z}_{2^{32}} \\
&\dots \\
x_{n-1} &\xleftarrow{\$} \mathbb{Z}_{2^{32}} \\
x_n &\leftarrow (x - \sum_{i=1}^{n-1} x_i) \bmod 2^{32}
\end{aligned}
$$

Then we have $\sum_{i=1}^{n} x_i = x$ and each share x_i is uniformly random, which means that the parties receiving the shares will learn nothing about the original secret. Notice that the additive scheme is an n-out-of-n secret sharing scheme, since all shares are required for reconstructing the data [5].

SMC protocols based on secret sharing require multiple computing parties since the data is distributed. To ensure data privacy, the computing parties must not learn the shares of the other parties, as otherwise they could reconstruct the secret-shared data. To that end, the protocol performs distributed secure computations on the shares. By using secret sharing, it is possible to perform computations in more diverse settings than the rather narrow two-party computation model.

In general, we can divide the actors of a secure multi-party computation into three groups. First, a number of *input parties* $\mathcal{IP}_i$ will provide the input to the computation. Then, *computing parties* $\mathcal{CP}_j$ will perform secure computations using some cryptographic protocol, and finally, the result is published to one or more result parties $\mathcal{RP}_k$.

The number of input parties and result parties may vary according to the context where secure computation is applied. The number of computing parties depends on the actual protocols that are used, since different protocols may require a different amount of parties to guarantee security. The additive secret sharing scheme, for example, requires at least three computing parties with an honest majority to remain information-theoretically secure [5].

Also, all three groups can overlap to some extent. For example, some input parties may also be result parties at the same time. In the case of two-party secure computation, both parties are input parties and computing parties, with at least one of them being a result party.

Let us now describe a general construction for a SMC protocol using secret sharing with one input party and one result party [5] (also illustrated in Figure 4).

1. An input party $\mathcal{IP}$ divides its data into n shares.
2. $\mathcal{IP}$ sends a share to each computing party $\mathcal{CP}_i$, $i = 1, \ldots, n$.
3. The computing parties $\mathcal{CP}_1$,...,$\mathcal{CP}_n$ perform secure computations on the shares.
4. The computing parties send the output shares to a result party $\mathcal{RP}$.
5. $\mathcal{RP}$ receives the shares and reconstructs the actual output.

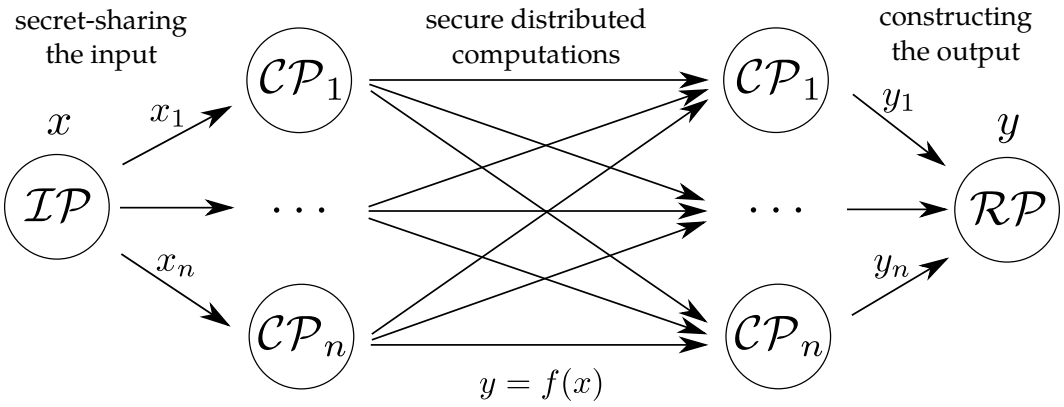

Figure 4: General construction of secret sharing protocols

The secret sharing protocol illustrated in Figure 4 calculates a specific function f, which is determined by the computations performed in step 3. To build a protocol that calculates a different function g, different secure computations must be specified. Constructing such protocols is not always straightforward, since it requires careful manipulation of the shares to provide the correct result while

ensuring data privacy. For example, constructing protocols which perform floating-point arithmetic on secret-shared data is quite a challenging task [25].

Since computations are carried out on shares which are usually elements of a large ring or field, it is more natural to build protocols on secret-shared data which correspond to evaluating an *arithmetic circuit*. In an arithmetic circuit, instead of bit values, wires can carry larger values, for example elements of a ring or field.

Although it is possible to carry out arbitrary computations using linear secret-sharing schemes [15, 8], operations that require bit-level manipulation of data are more cumbersome to build with secret sharing. Yao's protocol is a somewhat more natural solution for implementing such operations, since it computes directly with individual bits and reduces the communication and complexity overhead of having to share each bit individually. For simple arithmetic operations, however, secret sharing protocols are more efficient, as there is no need to regard the bit-level structure of data.

2.2.2 Homomorphic cryptography

We have explored the additive secret sharing scheme in the previous section. Notice that performing an addition operation on additively secret-shared data is very simple and requires no communication between computing parties. Namely, to compute the sum of two shared values x and y, each share-holder $\mathcal{CP}_i$ will simply add his shares together, receiving $z_i = x_i + y_i$. Then

$$\sum_{i=1}^{n} z_i = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i = x + y$$

which means we have succesfully performed addition without any communication between the computing parties.

This is called a *homomorphic* property of the secret sharing scheme. For additive secret sharing, addition is homomorphic as it can be performed by using only local computations. In general, a cryptographic encryption scheme is considered homomorphic, if there exists some efficient algorithm for calculating functions on encrypted inputs and receiving the output also in encrypted form, without actually decrypting the data.

More formally, let $\mathcal{E}(m)$ denote encrypting a plaintext m. If there exists an efficient algorithm $\mathcal{A}_f$ for some function f which computes

$$\mathcal{A}_f(\mathcal{E}(m_1), \ldots, \mathcal{E}(m_i)) = \mathcal{E}(f(m_1, \ldots, m_i)),$$

then $\mathcal{E}$ is a homomorphic encryption scheme in terms of f. Of course, $\mathcal{A}_f$ is not allowed to decrypt the ciphertexts directly, as this would defeat the privacy property of the scheme. Here we silently assume that $\mathcal{E}$ is semantically secure, that is, it is hard to learn anything about the plaintext by looking at the encrypted

ciphertext. Also, decryption with the secret key should produce the correct result, meaning that the original plaintext should be obtainable.

A *fully homomorphic* encryption scheme allows both addition and multiplication to be performed on ciphertexts without decrypting them, which in theory provides the possibility to evaluate any circuit. The theoretical feasibility of fully homomorphic encryption was first proven by C. Gentry [20]. His constructed encryption scheme relies on the hardness of some problems on integer lattices. However, there are also a number of schemes which are only partially homomorphic. For example, the Paillier cryptosystem [35] is only additively homomorphic, similarly to the additive secret sharing scheme.

A possible application of homomorphic cryptography is private information retrieval schemes, wherein a server can hold some client's encrypted data and respond to encrypted queries, without learning anything about the query or the data. Also, general SMC can be performed by using homomorpic cryptography [17].

However, providing full-scale SMC using only homomorphic cryptography is currently not practical performance-wise and is much more inefficient than other SMC methods. One of the first results of evaluating a real-life circuit using homomorphic encryption appeared in [21], where the authors evaluated an AES-128 circuit. The best results received were 36 hours for encrypting a single block, and 5 minutes of amortized time per block when using a different algorithm and running the computation for 65 hours.

This shows that for practical applications, homomorphic encryption by itself is currently not viable and other methods must be used. However, in many cases, homomorphic encryption can be used to enhance other SMC methods [16, 22, 17, 37]. In this thesis, we concentrate on the Yao's garbled circuits and secret sharing methods as they provide us with the necessary efficient tools to build our general-purpose SMC protocol.

3 Two compilers from C code to Boolean circuits

One important issue with the garbled circuits approach is that it requires circuit representations for the functions that are evaluated. Building a circuit that calculates a specific function is not a trivial task in itself, since a circuit representation for a single function can contain millions or even billions of gates. For example, the RSA-1024 circuit used in [29] contains 42 billion Boolean gates.

As Boolean circuits can be found extensively in hardware, one might reuse these for use in protocols based on garbled circuits. As Yao's protocol entails considerable performance overheads compared to ordinary computation, it is vital to use the most efficient circuit representations available. Measures for circuit efficiency however are slightly different for hardware circuits and circuits used for SMC. A small amount of gates and low depth of the circuit structure are desirable in both cases, but some specific constructions are more optimal for Yao's protocol. For example, using a well-known optimization, circuits with a large fraction of XOR-gates can be evaluated much more efficiently [26]. Also, existing hardware circuits might not contain the functions that would be useful to compute in a SMC setting, or such designs might be proprietary.

An automatic circuit generating tool would be very helpful for creating arbitary efficient circuits suitable for SMC applications. There have been some constructions in the past which aim for simplicity in generating circuits for secure protocols [34, 2, 22], but they offer only a limited number of existing building blocks for creating circuits and are mainly designed for use on a specific SMC platform.

In recent years, two notable state-of-the-art circuit compilers from C have emerged - CBMC-GC [23] and PCF (Portable Circuit Format) [27]. Both tools can compile C programs to a corresponding Boolean circuit representation and are optimized for use in garbled circuit protocols. The fact that the familiar C language is used makes them especially convenient tools, since no domain-specific knowledge is needed to generate general-purpose circuits which are usable in any SMC platform. Also, since C is a very widely used programming language, there is a large amount of existing code that could potentially be reused without having to program all the necessary computations from scratch.

Both compilers have already been used succesfully with different secure computation frameworks [24, 29]. We have also integrated PCF with the Sharemind platform [5] through the general-purpose secure computation protocol described in this thesis (see Section 4). Note that although both compilers are focused on the two-party computational model, this is not an inherent shortcoming as the circuits generated can still be used to compute the necessary function regardless of where the input is received from.

We will now describe both compilers in detail and provide an overview comparison in terms of their efficiency and functionality.

3.1 CBMC-GC

The CBMC-GC compiler for C [23] is based on the CBMC bound model checker [14], which allows the verification of ANSI C programs against different assertions. The CBMC model checker transforms the verified C program into a Boolean formula, which can then be analyzed for satisfiability. The produced Boolean formula contains a bit-precise representation of the program execution in memory, along with assertions that need to be verified. Program traces that violate the assertions can then be found using a Boolean satisfiability solver.

The CBMC model checker is useful for building a circuit compiler as it provides an exact bit-level representation of the program. The CBMC-GC compiler is actually a modified version of the model checker, which uses the Boolean formula that CBMC produces to translate the program into a circuit. CBMC-GC produces circuit descriptions in a straightforward textual format, which lists all gates of the circuit, specifying the functionality and connecting wires of each gate and also the mappings of external input and output wires to the program variables.

CBMC-GC places some restrictions on the programs it can succesfully compile. Since CBMC is a bounded model checker, it requires the analyzed program to terminate in a bounded number of steps. The bound is found either by static analysis of the code or by receiving it as user input. The bound is necessary, since a bit-precise representation of the program execution requires all loops and recursions to be fully unrolled and therefore, non-terminating programs will have an infinitely large execution description.

CBMC-GC also executes various methods to minimize and optimize the structure of the produced circuit, for details we refer to [23, 18]. The latest public version of CBMC-GC at the time of writing this thesis (v0.9.3) supports almost full ANSI C semantics, considering the bound restrictions. Different bit-length variables are supported in a single program, also pointer and floating point arithmetic. However, only scalar variables are allowed for input and output. Overall, the set of possible computations that are supported by CBMC-GC can be considered quite diverse for SMC purposes. However, due to the need of unrolling all loops in the program, compilation times can be expected to be quite long for larger circuits.

3.2 PCF

The PCF circuit compiler takes a fundamentally different approach to compiling circuits [27]. Instead of producing a full circuit representation, the PCF system uses a compact format which can be interpreted similarly to a stack machine. In contrast to CBMC-GC, loops and recursions in the compiled program are not unrolled, but rather, basic program control flow structures appear also in the final

circuit representation. Intuitively, the PCF format can be thought of as not an actual circuit, but a program which calculates the full circuit description at runtime. This method considerably reduces the amount of necessary RAM for evaluating such circuits and allows much larger circuits to be compiled in reasonable time, due to the fact that the circuit is not analyzed in a bit-precise manner, but using more high-level constructs. On the other hand, the programmer is left responsible for making sure that the written program will terminate, although some more obvious cases which can lead to infinite loops are detected by the compiler itself.

Overall, the PCF system consists of three parts. First, it uses a *front-end* compiler to translate the C program into a simpler intermediary representation. Currently, PCF uses the LCC compiler [31] to translate the program into a bytecode format [29]. The use of LCC bytecode is motivated by the fact that the bytecode representation and its possible optimizations are machine-independent. The authors have also suggested supporting LLVM bytecode in the future [28], so in theory, PCF could compile circuits from a number of different high-level programming languages.

The bytecode is then translated to the PCF format by the *back-end* compiler. The resulting circuit is then optimized with various methods to minimize the size of the circuit, for details we refer to [29]. To garbling and evaluating the circuit in PCF format, the PCF *interpreter* is used. The interpreter can be used as an external library by any SMC framework. The interpreter does not make any assumptions about the security model or garbling scheme used in the SMC system. The interpreter simply provides an interface which emits a circuit's gates one-by-one and allows data to be written to and read from the gate's input and output wires via customly definable callback functions. The actual circuit is therefore emitted only during actual evaluation, and is defined by the interpreter's inner state. The full circuit structure is never held in memory at once, since the interpreter will write over old wire pointers, if they are no longer used.

However, this approach of sequentially parsing the circuit at runtime removes the possibility to garble many gates in parallel, since the interpreter's state is changed with every emitted gate. For example, old wire values used in a previous gate may be overwritten by emitting the next gate. Therefore, it is necessary to garble and evaluate gates one-by-one. Also, since the whole circuit structure is not explicitly available, it is harder to analyze whether the actual circuit meets the conditions which are required for a certain SMC protocol. The need for precisely defining the circuit structure used in Yao's protocol is stressed in [1], since ambiguous defininitions of circuits may affect the correctness and security of the garbling scheme used.

Currently, PCF supports only 32-bit integer variables, which makes it harder to produce optimal circuits for operations which use smaller bit-length or 64-bit

data.

3.3 Comparison of two compilers

We performed a small number of experiments with both CBMC-GC and PCF to assess their performance. We used the publicly available CBMC-GC version 0.9.3 [12] and the PCF version used for the Kreuter et al. paper [29] we received from the authors themselves. All tests were performed on a workstation with 16 GB of RAM and an Intel® Core™ i7-870 2.93 GHz processor.

We tested compiling a few circuits used for benchmarking in [29] and [23]. Although we tried to use identical code with both compilers, some modifications to the compiled code were necessary. Namely, in CBMC-GC, we were forced to declare input and output variable explicitly, whereas in PCF, reading input and writing output are done using specific declared functions. In both cases, we copied the inputs to globally declared arrays with fixed length and performed computations using these arrays. Output was similarly written to a separate array. In PCF versions of the scripts, we read the input using a *for*-loop over the input size and reading 32 bits of the input in each iteration, since explicitly reading each input variable without using cycles was suboptimal for PCF due to the high cost of certain pointer operations [29].

We illustrate these differences with code examples. Code example 1 presents a C program which can be compiled using CBMC-GC. Note the variable names with INPUT and OUTPUT prefixes. In Code example 2, a PCF compatible code is presented. Both programs will result in an equivalent circuit being compiled.

```
int A[2], B[2], C[1];

void main(INPUT_A_0, INPUT_A_1, INPUT_B_0, INPUT_B_1) {
  A[0] = INPUT_A_0;
  A[1] = INPUT_A_1;
  B[0] = INPUT_B_0;
  B[1] = INPUT_B_0;
  C[0] = 0;
  int i;
  for (i = 0; i < 2; i++) {
    C[0] += A[i] + B[i];
  }
  int OUTPUT_0 = C[0];
}
```

Code example 1: CBMC-GC compatible C code

```
int alice(int); int bob(int); void output_alice(int);
int A[2], B[2], C[1];

void main() {
  int i;
  for (i = 0; i < 2; i++) {
    A[i] = alice(i*32);
    B[i] = bob(i*32);
  }
  C[0] = 0;
  for (i = 0; i < 2; i++) {
    C[0] += A[i] + B[i];
  }
  output_alice(C[0]);
}
```

Code example 2: PCF compatible C code

The C code for the Hamming distance and matrix multiplication was taken from benchmarking programs bundled with the CBMC-GC compiler and code for the RSA-256 circuit was bundled with PCF. The 128-bit sum implementation was self-written. We stress that although the compilers might produce more optimal circuits for a more cleverly written C algorithm, here our goal is to compare both compilers under similar conditions. The results of our experiments are presented in Tables 3 and 4 for CBMC-GC and PCF respectively.

Compilation times were calculated as the mean of 10 experiments except for the Hamming distance and matrix multiplication circuits compiled with CBMC-GC, where compilation times are the mean of 2 experiments. All experiments were run with enabling all of the compilers' optimizations. However, the matrix multiplication circuit was compiled with CBMC-GC with limiting the SAT-based minimization iterations to a maximum of 5, to considerably reduce the compilation time. For reference, compiling the 128-bit sum circuit used 27 SAT-minimization iterations and the Hamming distance used 38 iterations.

We note that in our experiments, we noticed that only a single core of the processor was used while compiling with either compiler, which suggests that considerable performance gains could be possible through parallelization.

In summary of our experimental results, we can see that CBMC-GC's compiling times are significantly larger than PCF's, although CBMC-GC does produce more optimized circuits in the end due to the smaller number of non-XOR gates in the circuit. However, large circuits like the RSA-256 take huge amounts of time to compile with CBMC-GC, but PCF scales exceptionally well with circuit size. For smaller circuits at least, it seems CBMC-GC is more capable at optimizing the circuit than PCF. However, as can be seen with the 5x5 matrix multiplication circuit, PCF is much faster in producing fairly optimized circuits. It is possible

Circuit	**Total gates**	**Non-XOR gates**	**Compilation time (s)**
128-bit sum	1,924	539	$85.18 \pm 0.99\%$
1600-bit Hamming dist.	16,606	4,038	∼3,323
32-bit 5x5 matrix mult.	395,550	148,650	∼15,815

Table 3: Experimental results of compiling circuits with CBMC-GC showing size of produced circuits and compilation time.

Circuit	**Total gates**	**Non-XOR gates**	**Compilation time (s)**
128-bit sum	4,100	1,403	$26.56 \pm 1.1\%$
1600-bit Hamming dist.	32,912	6,375	$18.44 \pm 0.60\%$
32-bit 5x5 matrix mult.	451,925	131,875	$159.00 \pm 0.30\%$
256-bit RSA	605,028,781	240,058,952	$103.68 \pm 0.22\%$

Table 4: Experimental results of compiling circuits with PCF showing size of produced circuits and compilation time.

that CBMC-GC would have produced a more optimal circuit if allowed to fully optimize, but in our test environment, it would have taken days to compile.

From this we can conclude that comparing the current state of both compilers, the CBMC-GC compiler might be more useful for generating small and moderately sized circuits, since memory consumption is not a concern with small circuits. Even compile times of several hours are acceptable from an end-user viewpoint, since compilation is a one-time offline process. It is more important that the produced circuit for a key primitive be as efficient as possible, since larger operations can then be built by composing primitives, for example. Also, since CBMC-GC supports 8-, 16-, 32- and 64-bit data types, one can implement certain functionality more easily and the circuit itself will be more optimal, as opposed to using only 32-bit integers.

On the other hand, PCF thrives with large circuits which can be expressed by cyclic or recursive properties. With these kinds of circuits, CBMC-GC becomes unpractical to use due to such high compiling times. Also, the circuit produced will be significantly large and the memory-consumption during the evaluation of such circuits becomes a serious issue. There are no such concerns with PCF circuits, however.

We will now describe a protocol which is capable of using Boolean circuits to perform secure multi-party computations in a generic computational model with any number of input and result parties.

4 Combining garbled circuits with secret sharing

We have now explored and compared the advantages of many useful SMC methods. Our main goal is to build a protocol which can leverage the capabilities of modern circuit compilers to easily build efficient SMC protocols for evaluating any Boolean function. We will call this the *hybrid protocol*, since it combines the efficiency of secret sharing with the robustness of Yao's garbled circuits approach. The hybrid protocol will work on bitwise secret-shared data, meaning that every bit of the data is shared independently. As an added gain, our protocol will be usable in more diverse settings than the regular two-party computation model. Much of the following material has also been published in a recent research report [41].

In the scope of this thesis, our goal is to provide security in the *passive* (*honest-but-curious*) model, where the corrupted party is assumed to be *honest* in the sense that it strictly follows the protocol, but *curious*, meaning that it will try to use all data that is available to learn something about other parties' private inputs. Security in the passive model guarantees that a corrupted computing party cannot learn anything else during a secure computation besides what can be deduced from its own input and output. A more evolved security model is the *active* model, where we would have to consider the possibility that one or more computing parties are maliciously tampering with the protocol by sending incorrect messages or not following the protocol in other ways.

We will give the description of our protocol in the Sharemind privacy-preserving computational platform setting, since our prototype implementation of the protocol is built directly in the Sharemind platform [40]. We chose Sharemind because it was available to us and provides a well-developed flexible computation environment built on secret sharing which could easily be extended with new protocols.

Sharemind provides a runtime which implements a number of different *protection domains* [7]. All data which belongs under one protection domain is guarded by the same set of algorithms and protocols for secure data storage and computations. Protection domains can use different methods and a varying number of computing parties to achieve the desired security goals. Our prototype naturally extends the additive 3-party passive (`additive3pp`) domain.

The `additive3pp` domain is based on the 3-out-of-3 additive secret sharing scheme. Therefore, it uses three computing parties to run secure distributed computations on secret-shared data. There are already a number primitive operations implemented in the `additive3pp` domain which are also available for us to use in our hybrid protocol implementation. All of these protocols provide security in the passive model and can tolerate at most one corrupted party [5].

We will denote the computing parties as $\mathcal{CP}_1$, $\mathcal{CP}_2$ and $\mathcal{CP}_3$. In the hybrid protocol, $\mathcal{CP}_1$ and $\mathcal{CP}_2$ respectively will take the roles of the garbler and evaluator from Yao's protocol. Since the hybrid protocol operates on secret-shared data, we

will use $\overline{[\![b]\!]} = [\![b_1, b_2, \ldots, b_n]\!]$ to denote a bit vector with length n which is shared between the computing parties, where $[\![b_i]\!]$ represents the i-th shared bit from the vector. Note that each bit b_i is shared individually.

Now, suppose we have a bit vector $\overline{[\![x]\!]} = [\![x_1, \ldots, x_n]\!]$ shared between the computing parties and we want to compute the result $f(\overline{[\![x]\!]}) = \overline{[\![y]\!]} = [\![y_1, \ldots, y_m]\!]$, where f is a function $f : \{0,1\}^n \to \{0,1\}^m$. Let us assume that all computing parties have access to a Boolean circuit C_f which calculates the function f. Since the garbling process produces a pair of tokens for each wire, we will use $X_j^b \in \{0,1\}^k$ to denote the token of the j-th wire corresponding to bit $b \in \{0,1\}$, where k is the length of the generated tokens. We say X_j^b has the *semantics* of b. The main parts of the hybrid protocol are illustrated in Figure 5.

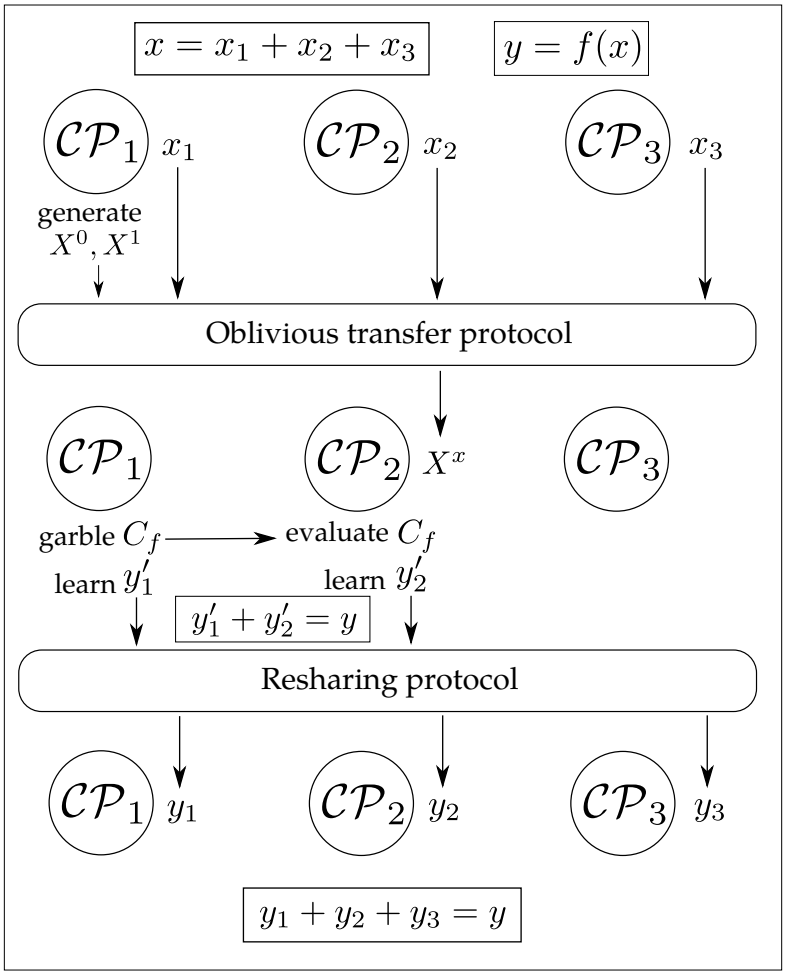

Figure 5: Hybrid protocol overview

Algorithm 1: Hybrid protocol for processing bitwise secret-shared data with a garbled circuit

Input: Shared bit vector $\overline{[\![x]\!]} = [\![x_1, \ldots, x_n]\!]$
Boolean circuit C_f for the function $f : \{0,1\}^n \to \{0,1\}^m$
Output: Shared bit vector $\overline{[\![y]\!]} = [\![y_1, \ldots, y_m]\!]$ such that $\overline{[\![y]\!]} = f(\overline{[\![x]\!]})$

1 **foreach** *input wire* $i \in \{1, \ldots, n\}$ **do**
2 $\quad \mathcal{CP}_1$ generates a token pair $(X_i^0, X_i^1) \in \{0,1\}^k \times \{0,1\}^k$
3 The computing parties initiate an oblivious transfer protocol which results in $\mathcal{CP}_2$ receiving $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$ (the input tokens corresponding to the actual input bits)
4 $\mathcal{CP}_1$ garbles circuit C_f and sends the garbled truth tables to $\mathcal{CP}_2$
5 $\mathcal{CP}_2$ evaluates garbled C_f using input tokens $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$ and receives output wires' tokens $\left\{X_{o_1}^{y_1}, \ldots, X_{o_m}^{y_m}\right\}$
6 The computing parties produce their output shares and reshare the output to receive $\overline{[\![y]\!]}$
7 **return** $\overline{[\![y]\!]}$

Note, that unlike the baseline Yao's garbled circuits protocol where input comes from both the garbler and evaluator, here the input is secret-shared between all computing parties as well as the output. For oblivious transfer between computing parties, the protocol makes use of Sharemind's secure multiplication and addition protocols on secret-shared integers to perform an oblivious choice [8].

Since we do not want any of the computing parties to actually learn the output of the calculation, the external output wires' values are also encrypted by the garbler. However, the evaluator will not send the result of the evaluation back to the garbler, namely the external output tokens. Instead, the computing parties will calculate random shares of the output using a *perfectly secure resharing protocol* [5]. For clarity, a high-level algorithm of the hybrid protocol is presented as Algorithm 1.

4.1 Parts of the hybrid protocol

The next sections will cover different parts of the hybrid protocol in detail. We will also present a proof of the *perfect simulatability* of the oblivious transfer protocol following the blueprint described in [5] and argue that the whole hybrid protocol is secure against a computationally bounded adversary.

Algorithm 2: Oblivious transfer of input tokens

Input: $\mathcal{CP}_1$ holds the input tokens $\{X_1^0, \ldots, X_n^0, X_1^1, \ldots, X_n^1\}$
The input bit vector $\overline{[\![x]\!]} = [\![x_1, \ldots, x_n]\!]$ is shared between all parties
Output: $\mathcal{CP}_2$ receives input tokens $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$

1 $\overline{[\![X^0]\!]} = [\![X_1^0, \ldots, X_n^0]\!]$ and $\overline{[\![X^1]\!]} = [\![X_1^1, \ldots, X_n^1]\!]$ are instantiated as shared values, with $\mathcal{CP}_1$ taking as his shares the actual tokens and $\mathcal{CP}_2$, $\mathcal{CP}_3$ taking zero-shares
2 $\overline{[\![\hat{x}]\!]} = \overline{[\![1]\!]} - \overline{[\![x]\!]}$
3 $\overline{[\![Y']\!]} = \overline{[\![X^0]\!]} \cdot \overline{[\![\hat{x}]\!]}$
4 $\overline{[\![Y'']\!]} = \overline{[\![X^1]\!]} \cdot \overline{[\![x]\!]}$
5 $\overline{[\![Y]\!]} = \overline{[\![Y']\!]} + \overline{[\![Y'']\!]}$
6 $\overline{[\![Y]\!]}$ is declassified to $\mathcal{CP}_2$ as $\mathcal{CP}_1$ and $\mathcal{CP}_3$ send their shares of $\overline{[\![Y]\!]}$ to $\mathcal{CP}_2$
7 $\mathcal{CP}_2$ combines the shares of $\overline{[\![Y]\!]}$ to get $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$
8 **return** $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$

4.1.1 Oblivious transfer

Let us first consider the oblivious transfer of the input tokens. The input to the protocol is $[\![x_1, \ldots, x_n]\!]$ and the garbler $\mathcal{CP}_1$ has generated corresponding input tokens $\{X_1^0, \ldots, X_n^0, X_1^1, \ldots, X_n^1\}$. We now need an oblivious transfer protocol from $\mathcal{CP}_1$ to $\mathcal{CP}_2$ which satisfies the following conditions:

1. As the result, $\mathcal{CP}_2$ should learn $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$ and nothing else
2. $\mathcal{CP}_1$ must not learn $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$, i.e, which tokens were transferred to $\mathcal{CP}_2$
3. No computing party can learn the input $\{x_1, \ldots, x_n\}$

It can be seen easily that if we had an oblivious choice protocol which follows conditions 2, 3 and outputs $[\![X_1^{x_1}, \ldots, X_n^{x_n}]\!]$ shared between the computing parties, then satisfying condition 1 is trivial, since we can extend the oblivious choice by simply sending all result shares from other computing parties to $\mathcal{CP}_2$. Performing an oblivious choice on secret-shared data, however, can be easily implemented using secure multiplication and addition protocols. The resulting oblivious transfer protocol is described in Algorithm 2.

On lines 2-5, Sharemind's `additive3pp` protection domain multiplication and addition protocols are used to perform an oblivious choice. The calculations can be summarized as $\overline{[\![Y]\!]} = \overline{[\![X^0]\!]} \cdot (\overline{[\![1]\!]} - \overline{[\![x]\!]}) + \overline{[\![X^1]\!]} \cdot \overline{[\![x]\!]}$, but for clarity, separate primitive operations which are implemented by different protocols are written on separate lines. As the result, $\overline{[\![Y]\!]}$ contains the necessary tokens which need to be

transferred to $\mathcal{CP}_2$. Then on line 6, the shares of $\overline{[\![Y]\!]}$ are sent to $\mathcal{CP}_2$ who can combine them to receive the actual input tokens. Note that the tokens X^i are bit strings of length k. Although the multiplication and addition protocols are defined on elements of a ring $\mathbb{Z}_{2^n}$, we can easily encode the tokens as an array of $\mathbb{Z}_{2^n}$ elements when $k \bmod n = 0$, and extend $\overline{[\![x]\!]}$ and $\overline{[\![\hat{x}]\!]}$ to match the extended length of the tokens.

We will prove in Section 4.2.1 that the oblivious transfer protocol described in Algorithm 2 is perfectly simulatable according to [5].

4.1.2 Garbling

After the oblivious transfer of the input tokens to the evaluator, the hybrid protocol is very similar to the standard Yao's protocol and can be implemented using various garbling schemes and circuit formats.

Our implementation uses the garbling scheme GaXR presented by Bellare et al. [1], which is based on modeling fixed-key AES as a random permutation. Bellare et al.'s garbling scheme is one of the most efficient garbling schemes to date and takes full advantage of hardware with AES-NI support. Also, the chosen scheme is compatible with the well-known *free-XOR* [26] and *garbled row reduction* [36] optimizations. For encryption, we chose the A4 construction over other alternatives presented in the paper since it helps reduce network communication, which we expected to be a performance bottleneck. Note that this garbling scheme is defined for a specific circuit construction. The circuit must consist of only 2-to-1 gates with arbitrary fan-out and functionality. Each wire, which is not an external input wire, must be an outgoing wire of a gate. External output wires cannot be external input wires or inputs to gates. The circuit's external input and output wires must be unique and no wire can twice feed a gate.

Following the notation of [1], each circuit C can be described as a tuple (n, m, q, A, B, G), where n is the number of external input wires, m the number of external output wires and q the number of gates in C. Then $InputWires = [1, \ldots, n]$, $Wires = [1, \ldots, n+q]$, $OutputWires = [n+q-m+1, \ldots, n+q]$ and $Gates = [n+1, \ldots, n+q]$. Functions $A : Gates \to Wires \backslash OutputWires$ and $B : Gates \to Wires \backslash OutputWires$ respectively identify the first and second input wire of any gate. The function $G : Gates \times \{0,1\}^2 \to \{0,1\}$ determines the functionality of each gate. For a given $g \in Gates$, the function $G(g) : \{0,1\}^2 \to \{0,1\}$ denotes the functionality of gate g. We have presented detailed algorithms of the whole protocol for each computing party in Figure 6.

For each input wire $i \in InputWires$, the garbler $\mathcal{CP}_1$ generates a token pair (X_i^0, X_i^1) with X_i^0 and X_i^1 having the semantics of 0 and 1 respectively. We will call the last bit of a wire token it's *value bit*, since the evaluator uses it to choose which row in the garbled truth table to decrypt. However, to hide the true se-

Algorithm 3: Hybrid protocol algorithm of $\mathcal{CP}_1$

Input: Input shares $x_{*1} = [x_{11}, \ldots, x_{n1}]$ and circuit $C_f = (n, m, q, A, B, G)$

Output: Shares $[y_{11}, \ldots, y_{m1}]$ of $\overline{[\![y]\!]}$ such that $\overline{[\![y]\!]} = f(\overline{[\![x]\!]})$

$R \xleftarrow{\$} \{0,1\}^{k-1} \parallel 1$
for $i \leftarrow 1$ **to** n **do**
  $p_i \xleftarrow{\$} \{0,1\}$
  $X_i^0 \xleftarrow{\$} \{0,1\}^{k-1} \parallel p_i, \quad X_i^1 \leftarrow X_i^0 \oplus R$
$\mathtt{OT}\ ([X_1^0, \ldots, X_n^0], [X_1^1, \ldots, X_n^1], x_{*1})$
for $g \leftarrow n+1$ **to** $n+q$ **do**
  $a \leftarrow A(g), \quad b \leftarrow B(g)$
  if $G(g) = XOR$ **then**
    $X_g^0 \leftarrow X_a^0 \oplus X_b^0, \quad X_g^1 \leftarrow X_g^0 \oplus R$
  else
    for $i \leftarrow 0$ **to** $1, j \leftarrow 0$ **to** 1 **do**
      $u \leftarrow i \oplus \mathtt{lsb}(X_a^0)$
      $v \leftarrow j \oplus \mathtt{lsb}(X_b^0)$
      $r \leftarrow G(g, u, v)$
      if $i = 0$ **and** $j = 0$ **then**
        $X_g^r \leftarrow \mathtt{Enc}(X_a^u, X_b^v, g, 0^k)$
        $X_g^{r-1} \leftarrow X_g^r \oplus R$
      else
        $P[g, i, j] \leftarrow$
          $\mathtt{Enc}(X_a^u, X_b^v, g, X_g^r)$
Send P to $\mathcal{CP}_2$
for $i \leftarrow 1$ **to** m **do**
  $y'_{i1} \leftarrow \mathtt{lsb}(X^0_{n+q-m+i})$
$[y_{11}, \ldots, y_{m1}] \leftarrow \mathtt{Reshare}([y'_{11}, \ldots, y'_{m1}])$
return $[y_{11}, \ldots, y_{m1}]$

Algorithm 4: Hybrid protocol algorithm of $\mathcal{CP}_2$

Input: Input shares $x_{*2} = [x_{12}, \ldots, x_{n2}]$ and circuit $C_f = (n, m, q, A, B, G)$

Output: Shares $[y_{12}, \ldots, y_{m2}]$ of $\overline{[\![y]\!]}$ such that $\overline{[\![y]\!]} = f(\overline{[\![x]\!]})$

$[X_1, \ldots, X_n] \leftarrow \mathtt{OT}\ (0^{k \cdot n}, 0^{k \cdot n}, x_{*2})$
Receive P from $\mathcal{CP}_1$
for $g \leftarrow n+1$ **to** $n+q$ **do**
  $a \leftarrow A(g), \quad b \leftarrow B(g)$
  $i \leftarrow \mathtt{lsb}(X_a), \quad j \leftarrow \mathtt{lsb}(X_b)$
  if $G(g) = XOR$ **then**
    $X_g \leftarrow X_a \oplus X_b$
  else if $i = 0$ **and** $j = 0$ **then**
    $X_g \leftarrow \mathtt{Enc}(X_a, X_b, g, 0^k)$
  else
    $X_g \leftarrow \mathtt{Dec}(X_a, X_b, g, P[g, i, j])$
for $i \leftarrow 1$ **to** m **do**
  $y'_{i2} \leftarrow \mathtt{lsb}(X_{n+q-m+i})$
$[y_{12}, \ldots, y_{m2}] \leftarrow \mathtt{Reshare}([y'_{12}, \ldots, y'_{m2}])$
return $[y_{12}, \ldots, y_{m2}]$

Algorithm 5: Hybrid protocol algorithm of $\mathcal{CP}_3$

Input: Input shares $x_{*3} = [x_{13}, \ldots, x_{n3}]$

Output: Shares $[y_{13}, \ldots, y_{m3}]$ of $\overline{[\![y]\!]}$ such that $\overline{[\![y]\!]} = f(\overline{[\![x]\!]})$

$\mathtt{OT}\ (0^{k \cdot n}, 0^{k \cdot n}, x_{*3})$
$[y'_{13}, \ldots, y'_{m3}] \leftarrow 0^m$
$[y_{13}, \ldots, y_{m3}] \leftarrow \mathtt{Reshare}([y'_{13}, \ldots, y'_{m3}])$
return $[y_{13}, \ldots, y_{m3}]$

Figure 6: Detailed algorithms of the hybrid protocol for all computing parties.

mantics of the tokens from the evaluator, each token's value bit is masked with a random bit $p_i \xleftarrow{\$} \{0,1\}$, which is called a *permutation bit*. This means that if the permutation bit p_i equals 1, then the token's value bit will not correspond to the token's semantics, but rather the reverse. We will denote a token X_i^b with a value bit w_i as $X_i^b|w_i$. Finally, the generation results in tokens $(X_i^0|p_i, X_i^1|\overline{p_i})$ for wire i, with value bits p_i and $\overline{p_i}$ respectively. Note that the value bits may correspond to the tokens' semantics, or they may be flipped with probability $\frac{1}{2}$.

Since we are using the free-XOR technique, the input tokens are generated using a global random token $R \xleftarrow{\$} \{0,1\}^{k-1} \parallel 1$, where the last bit of R is always 1. This guarantees that tokens with different semantics also have different value bits, since X_i^1 is generated as $X_i^1 \leftarrow X_i^0 \oplus R$. The free-XOR technique enables the evaluator to evaluate XOR-gates without using the gate's garbled truth tables.

After the input tokens have been generated, each computing party participates in the oblivious transfer of the input tokens to $\mathcal{CP}_2$, using the protocol described in the previous section. We use `OT` to denote the call to the oblivious transfer protocol. Each party inputs their shares of $\overline{[\![x]\!]}$ to the `OT` protocol and $\mathcal{CP}_1$ also inputs the generated input tokens. Note that the `OT` protocol is run synchronously on all three computing parties.

After completing the oblivious transfer, $\mathcal{CP}_1$ starts garbling the circuit and produces the encrypted truth tables of all gates, which are saved in a data structure P. For each gate g, the encrypted output corresponding to input tokens with value bits $i, j \in \{0,1\}$ is stored in $P[g,i,j]$. However, due to the free-XOR technique, XOR-gates are not encrypted, but instead the tokens for a XOR-gate's output wire are chosen such that the evaluator $\mathcal{CP}_2$ need only perform a bitwise XOR operation on the two input tokens to receive the corresponding output token [26].

For non-XOR gates, the garbled row reduction technique applies, meaning that the first row of each non-XOR gate's truth table is not encrypted, but rather, the gate's output tokens are chosen such that $\mathcal{CP}_2$ can obtain the output token corresponding to input tokens $X_a^u|0$ and $X_b^v|0$ directly from those input tokens [36]. For the remaining three rows of the truth table, $\mathcal{CP}_1$ encrypts the corresponding output tokens using the input wires' tokens and saves them in P. The procedure $\texttt{lsb}(X)$ denotes taking the least significant bit from X and is used to extract the value bit from a wire's token.

After all gates are garbled, $\mathcal{CP}_1$ sends P to $\mathcal{CP}_2$, who will start evaluating the circuit gate-by-gate. The encryption scheme used to encrypt the truth tables is a function $\texttt{Enc} : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^\tau \times \{0,1\}^k \to \{0,1\}^k$ which takes secret tokens A and B and a tweak T to encrypt X, resulting in a ciphertext $\texttt{Enc}(A,B,T,X)$. The function `Enc` corresponds directly to the dual-key cipher construction A4 in Bellare et al.'s paper [1] and is defined as

$$\texttt{Enc}(A, B, T, X) = \pi(K \parallel T)_{[1:k]} \oplus K \oplus X$$

with

$$K = 2A \oplus 4B$$

where $X, A, B \in \{0,1\}^k$ and $T \in \{0,1\}^\tau$. The function $\pi : \{0,1\}^{k+\tau} \to \{0,1\}^{k+\tau}$ is a random permutation and $\pi(K \parallel T)_{[1:k]}$ denotes taking the first k bits of the result. In our implementation we use a fixed-key AES-128 with $k = 80$ and $\tau = 48$ to instantiate π, which provides reasonable security guarantees for this garbling scheme [1]. The encryption key for AES is randomly generated and renewed after each circuit evaluation. For the tweak T, we use the gate's index encoded as a 48-bit integer. $2A$ denotes a doubling function which can be implemented in many different ways, each providing slightly different security guarantees [1]. We chose *multiplication over finite field* $GF(2^k)$ due to it providing the best security guarantees. We used the irreducible polynomial $x^{80} + x^9 + x^4 + x^2 + 1$ from [38] to implement the finite field multiplication.

Decryption is symmetric and is defined as

$$\texttt{Dec}(A, B, T, X) = \texttt{Enc}(A, B, T, X).$$

After the evaluator $\mathcal{CP}_2$ has received both the external input tokens and the garbled truth tables, it will start evaluating the circuit gate-by-gate. For a non-XOR gate g with input wires a and b, the evaluator takes the value bits of the input tokens $X_a^i|w_a$ and $X_b^j|w_b$ and decrypts the truth table row $P[g, w_a, w_b]$ using $\texttt{Dec}(X_a^i, X_b^j, g, P[g, w_a, w_b])$. The decrypted output wire token will then be used to decrypt the truth tables of successive gates.

After $\mathcal{CP}_2$ has succesfully evaluated the whole circuit and received the external output tokens, the result $\overline{[\![y]\!]}$ is being shared between $\mathcal{CP}_1$ and $\mathcal{CP}_2$ as $\mathcal{CP}_2$ holds the value bits of the output tokens and $\mathcal{CP}_1$ holds the permutation bits. The XOR of the two provides the actual result since the value bits represent the semantics of the tokens, but are masked with the permutation bits.

The last step in the protocol is resharing the output securely between all three computing parties. This is done by using the `Reshare` protocol described in [5]. The `Reshare` protocol is also run synchronously on all three computing parties. Finally, the protocol ends with all computing parties holding a random share of the final result $\overline{[\![y]\!]} = f(\overline{[\![x]\!]})$.

4.2 Security proof sketch

We have now described the hybrid protocol in detail and will give a proof that the presented protocol is secure. Our goal is to prove that the hybrid protocol is secure

against a computationally bounded adversary. We will show that the oblivious transfer protocol is *perfectly simulatable* following the security proof framework of [5]. Note that a more evolved security framework has recently been published by Bogdanov et al. [6], which extends the framework of [5] to provide *universal composability* for simulatable protocols. We will also provide a proof sketch that the garbling procedure is secure, based on results from [1].

The security framework of [5] is based on the ideal vs real world paradigm. To prove the security of a multi-party computation protocol, we first model an ideal implementation of the protocol using a trusted third party. In the ideal implementation, the computing parties will submit their inputs to the trusted third party, who will perform all necessary computations securely and send the output back to the computing parties.

The aim is to show then that attacks against the protocol in the real world can be transformed into attacks in the ideal world, which are thereby roughly equivalent in terms of resources used and probability of success. Since we are working in the passive model, we will assume one of the computing parties is corrupted by an adversary $\mathcal{A}$, meaning that all inputs, outputs and internal state of that computing party will be seen by $\mathcal{A}$. If we can show that, for the protocol in question, the adversary cannot distinguish between the real and ideal world situations, then we can say that all attacks in both settings are roughly equivalent. This can be shown by constructing a *simulator* $\mathcal{S}$ which acts as a proxy between the trusted third party and the corrupted computing party. The simulator must be able to simulate all messages to the corrupted party that would occur in a real world run of the protocol, while being in the ideal world situation.

We will show for the oblivious transfer, that for each computing party $\mathcal{CP}_i$, there exists an efficient perfect non-rewinding simulator which can simulate all the incoming messages to $\mathcal{CP}_i$. A simulator is considered *perfect* if the distributions of the incoming messages to the adversary coincide in the real world and ideal world. A *non-rewinding* simulator does not rewind the adversary's state, rather, the protocol is executed in a straight line. To show perfect simulatability, we must also show that the corrupted party's output distribution is equal in the real world and ideal world situations. Perfect simulatability guarantees that the corrupted party does not learn anything except what can be derived from his own input and output.

We will also use Theorem 4 from [5], which states that a composition of several perfectly simulatable sub-protocols is also perfectly simulatable, if

- the output of each sub-protocol is either the input of another sub-protocol or the output of the main protocol, and
- the data dependency graph of sub-protocols is a directed acyclic graph.

In our analysis, we can ignore the resharing of the output shares in the end of the protocol, since we are using the perfectly secure resharing protocol from [5]. Intuitively, resharing the output guarantees that the output shares are completely independent of the input shares or any intermediate results, which ensures the composability of our protocol.

We will now show perfect simulatability for the oblivious transfer and security for the whole hybrid protocol. We argue, that combined with the recent work of [6], this security analysis can be extended to provide *universal composability* in the context of Canetti's universal composability framework [11]. However, this bachelor's thesis does not contain a full rigorous analysis of that claim.

4.2.1 Simulatability of oblivious transfer

Let us show that the oblivious transfer protocol described in Section 4.1.1 is perfectly simulatable.

The oblivious transfer can be divided into two parts:

1. the oblivious choice $\overline{[\![Y]\!]} = \overline{[\![X^0]\!]} \cdot (\overline{[\![1]\!]} - \overline{[\![x]\!]}) + \overline{[\![X^1]\!]} \cdot \overline{[\![x]\!]}$, and
2. declassifying $\overline{[\![Y]\!]}$ to $\mathcal{CP}_2$.

The first step is the oblivious choice, after which, the parties will receive shares (Y_1, Y_2, Y_3) which combine into Y. Notice that the oblivious choice consists of only the secure multiplication and addition protocols from [8], the composition of which is provably perfectly simulatable, since each computing party's incoming view consists of only uniformly random messages. Using Theorem 4 from [5] and the fact that the composition of the multiplication and addition protocols is perfectly simulatable, it follows that the whole oblivious choice is perfectly simulatable. Therefore, we know that a succesful simulator for the oblivious choice exists for whichever computing party and we can provide simulations Y_1°, Y_2° and Y_3° to the computing parties.

For computing parties $\mathcal{CP}_1$ and $\mathcal{CP}_3$, the only incoming communication that occurs during the whole protocol, is during the oblivious choice step. Therefore, the oblivious choice simulator is also a simulator for the whole oblivious transfer protocol for parties $\mathcal{CP}_1$ and $\mathcal{CP}_3$.

For $\mathcal{CP}_2$, however, we must additionally simulate the shares of $\overline{[\![Y]\!]}$ sent by $\mathcal{CP}_1$ and $\mathcal{CP}_3$. The shares Y_1 and Y_3 are also trivial to simulate, since the resharing of shares at the end of the multiplication protocol guarantees that the resulting output shares are uniformly distributed [5]. Therefore, the shares Y_1 and Y_3 are simulatable by simply sending random Y_1° and Y_3° to $\mathcal{CP}_2$. Finally, $\mathcal{CP}_2$ can construct some uniformly random Y° from the simulated shares, which is indistinguishable from an actual input token, since tokens generated by the garbler in the real world are also uniformly random.

We have therefore shown that an efficient simulator $\mathcal{S}_{OT}$ exists, which can simulate the real world execution of the oblivious transfer protocol. Since the distribution of all messages is identical to the real world, this proves the perfect simulatability of the oblivious transfer.

4.2.2 Security of the hybrid protocol

We will now provide a proof sketch that the whole hybrid protocol as described in Figure 6 is secure against a computationally bounded adversary. We can use our previous construction of $\mathcal{S}_{OT}$ to construct a simulator $\mathcal{S}$ for the whole protocol, ignoring the reshare in the end.

Here we cannot provide information-theoretic simulatability, since the circuit's garbled tables are not simulatable in that sense. An adversary with unbounded computational capability could learn something about the semantics of the input tokens based on the garbled tables. However, using a sufficiently secure garbling scheme, we can guarantee that for a real-life adversary with practical limitations, this probability is negligibly small.

We can see from the detailed algorithm of the whole hybrid protocol on Figure 6 that the only communication that occurs between the computing parties after the oblivious transfer, is sending the circuit's garbled tables P to $\mathcal{CP}_2$. Since there is no incoming communication to $\mathcal{CP}_1$ and $\mathcal{CP}_3$, we only need to concern ourselves with the incoming view of $\mathcal{CP}_2$. It is necessary therefore to simulate the garbled truth tables P, so that an adversary could not distinguish between the simulated tables and the actual garbled circuit in the real world.

Let us recall, that as the final result of the protocol, disregarding the reshare in the end, the output is secret-shared between $\mathcal{CP}_1$ and $\mathcal{CP}_2$. The permutation bits for the output tokens are held by $\mathcal{CP}_1$ and $\mathcal{CP}_2$ holds the value bits, which together form the actual semantics of the tokens.

We will construct the simulation as follows. During the oblivious transfer, the simulator $\mathcal{S}_{OT}$ has constructed messages Y_1°, Y_2° and Y_3°, which together combine into some input token Y°, that the evaluator will use to evaluate the garbled circuit sent to it.

The simulator can then use the same algorithm, as the garbler uses in the real protocol, to generate a set of garbled tables. We claim that the simulator can produce the garbled tables in such a way, that they correspond to the topological structure of the circuit being evaluated, and that the semantics of all output keys that the adversary will receive after evaluating using Y° are 0. However, the actual output shares the adversary receives, namely the value bits of the output tokens, are completely random. This is because during the garbling process, the actual semantics of the keys are masked with random permutation bits. The simulator follows the same process, therefore, as the semantics of the resulting output keys

are guaranteed to be 0, then the adversary will receive the generated random permutation bits as his share of the result.

Intuitively, the simulator can construct such tables, since it can modify the truth tables of the actual circuit and choose the semantics of the input tokens in a meaningful way, knowing also that the adversary will use the tokens Y° for evaluation. In this situation, the adversary learns nothing from the result of the evaluation, since the result is simply a random bit vector. We can then bound the adversary's chance of distinguishing between the simulated and real garbled tables based on a similar security analysis as can be found in both [1] and [43]. A rigorous proof of this would be highly technical, and as such, is left out of the scope of this thesis. However, since here the adversary does not possess any additional advantages than described in [1], we claim that reasonable security bounds can be proven.

5 Implementation details

We have built a prototype of the hybrid protocol to test its capability and usefulness. We built our prototype directly onto version 3 of the Sharemind privacy-preserving computational platform [5]. The implementation is written in C++ so that it could be seamlessly integrated into the protocol suite of Sharemind's `additive3pp` protection domain and achieve performance usable in practice. The prototype provides a separate primitive protocol among other secure computation protocols in the protection domain, like addition and multiplication, which we have also used in the hybrid protocol.

Using the common Sharemind development framework, we can send secret-shared data to the Sharemind servers and into our protocol through a client application. The output will eventually be shared between the servers, but we can receive the actual result by publishing it to the client application. Considering the general SMC application model, the client acts as an input and result party at the same time and the Sharemind servers act as computing parties. During the whole process, no single Sharemind server will learn anything about the inputs or outputs of the protocol.

5.1 Using the protocol to implement complex primitive operations

For a platform like Sharemind, our protocol prototype is well-suited for implementing primitive operations that are difficult or inefficient to express purely on the algebraic properties of secret sharing.

Current Sharemind protocols in the `additive3pp` domain are designed to operate on elements of a ring $\mathbb{Z}_{2^n}$ [8], but such protocols are not well-suited for robust bit-level manipulation over data types with an arbitrary bit-width. The hybrid protocol is much better suited for implementing such operations, since we have designed it to operate on bitwise secret-shared boolean vectors with an arbitrary length.

Since the hybrid protocol is composable, it can be combined with other primitive protocols to provide a versatile set of available secure operations to be used in applications. The output of one circuit can be used as the input to another circuit or even a different Sharemind protocol. The advantage of this approach is that we can support a wide range of possible secure calculations by composing primitives without having to generate circuits on the fly or compile huge Boolean circuits that include large input databases.

5.2 Circuit setup

Currently, we have used different circuits from Stefan Tillich and Nigel Smart from the University of Bristol [42] and Kreuter et al. [29] for testing and benchmarking our prototype. Also, we have experimented with the PCF circuit compiler and interpreter [27] to generate custom circuits from C programs and evaluate them using our hybrid protocol. In all cases, the circuits are stored as individual files on the Sharemind computing party servers. The protocol takes the name of the circuit as an input argument and parses the corresponding circuit file to compute the result. Both the garbler and evaluator parse the circuit exactly the same way since we are not required to hide the function that is evaluated, only the input and output data along with intermediary computation results.

Tillich and Smart's circuits are presented in a straight-forward format which lists all gates in the circuit along with their input and output wires in a topological order. For these circuits, we wrote our own simple circuit parser which reads the whole circuit structure into memory once before evaluation. Since our chosen garbling scheme is not suitable for garbling 1-to-1 gates, our parser optimizes out the logical inverse gates from these circuits. This is achieved by modifying the truth tables of the gates which have inputs originating from an inverse gate. However, since we do not want to lose XOR gates from the circuit, we do not modify their truth tables in case of an inverted input, but change the truth tables of successive non-XOR gates instead. Removing the inverse gates in such a way was not possible in all cases without violating the circuit structure needed for our garbling scheme. We therefore did not use these circuits to evaluate our protocol. Since evaluating the circuit requires no extra information from the circuit file, the parsing can be done in an offline phase, which reduces the time spent for garbling and evaluating. However, keeping the whole circuit directly in memory consumes large amounts of memory, which makes this method impractical for very large circuits.

Kreuter et al. kindly provided us with a parser for their circuits, which we used as basis in our protocol. Their circuits are presented in a compact binary format. The parser provides an interface which emits the circuit's gates one-by-one. At any time, only a working set of the circuit's wires are kept directly in memory. To efficiently parse the circuit at runtime, the whole circuit file is memory-mapped. However, this is simply a convenient method for reading the circuit file fast. In theory, the parser could hold only a fixed portion of the circuit file in memory at any one time. This method would scale much better with larger circuits in terms of memory consumption, but introduces a slight performance drawback, since circuits need to be parsed again for each successive evaluation, increasing the time spent for garbling and evaluating.

We also integrated the PCF interpreter succesfully with our hybrid protocol prototype. The C implementation of the PCF interpreter can be used as an ex-

ternal library and provides a circuit parsing black box to be used with any secure computation system to handle parsing and evaluating circuits compiled with the PCF compiler.

5.3 Optimizations

Our hybrid protocol contains some standard Yao protocol optimizations for the garbling and evaluating of circuits. As discussed in Section 4.1.2, the garbling scheme we use incorporates the *free-XOR* [26] and *garbled row reduction* [36] optimizations.

The *free-XOR* technique removes the need to garble XOR-gates by choosing the wire tokens in a clever way, which allows the evaluator to compute the correct output token directly from the input tokens. This greatly reduces the communication cost of the garbling procedure, as no garbled tables need to be sent for XOR-gates. Also, the computational cost for garbling XOR-gates is minimized, since there is no need to encrypt XOR-gates' truth tables. In theory, both communication and computational cost of garbling and evaluation are reduced to the fraction of non-XOR gates in the used circuit.

The *garbled row reduction* method further reduces the communication overhead of garbling by 25%. This is achieved by setting one of the tokens for the output wire of a non-XOR gate as a function of two input wire's tokens. Then for these two input tokens, the corresponding output token can be calculated directly by the evaluator without using the garbled truth table of the gate, effectively reducing the size of the garbled truth table by one row.

We are also using a *streaming* approach to the garbling and evaluating of circuits to parallelize the work of the garbler and the evaluator. When the garbler has finished garbling a batch of the circuit's gates, he can send the garbled truth tables to the evaluator, who can start evaluating the circuit while the garbler encrypts the next batch of gates. Currently, the batch size is fixed in our prototype, but in theory, the optimal batch size for circuit streaming can be fine-tuned to match the running Sharemind instance's network and hardware capabilities and the circuit being evaluated.

6 Experimental results

We now present the performance results of our implemented hybrid protocol prototype on Sharemind 3. The prototype was benchmarked with various circuits of different size from Stefan Tillich and Nigel Smart [42] and Kreuter et al. [29] The descriptions of the used circuits and the corresponding performance results are presented in Tables 5 and 6 respectively.

Table 5 lists - for every circuit - the size of the input in bits, the overall number of logic gates in the circuit, and the number of non-XOR gates in the circuit. We did not count logical inverse gates in the gate counts since these are optimized out for Tillich and Smart's circuits and Kreuter et al.'s circuits did not contain any. The batch size for streaming garbled tables to the evaluator was fixed for all test runs on 35,000 non-XOR gates' tables.

The tests were run on a cluster with three nodes, all hosting Sharemind version 3 servers, with each node acting as a single computing party. All nodes had 48GB of RAM and a 12-core 3GHz CPU which supports HyperThreading. The nodes were connected to a LAN with 1 Gbps full duplex links.

In Table 6, the performance results for all circuits are presented. The table lists the mean times spent on different phases of the protocol separately, and also the mean total elapsed time. All times are reported in milliseconds with a 95% confidence interval, where 123k denotes 123,000 ms. For each circuit we conducted two sets of experiments, one with AES-NI instructions enabled and the other without AES-NI, but using OpenSSL's v1.0.1 AES implementation instead. This enabled us to measure the effect of AES hardware instructions on our protocol's performance.

The initial parsing time for Kreuter et al.'s circuits is very small due to the fact that the circuit is actually parsed gate-by-gate during garbling and evaluation. Initially, the circuit is memory-mapped so that it could be efficiently parsed during runtime. The runtime parsing time is reflected in the garbling and evaluation times for Kreuter et al.'s circuits.

For Tillich and Smart's circuits, the whole circuit is parsed once before garbling/evaluation. The circuit format is much less efficient to parse than Kreuter et al.'s and therefore the initial parsing times are quite significant. Also, our custom-built parser might not be optimal for parsing. However, in theory the circuit could be parsed once in an offline phase and reused for multiple evaluations, but this approach is not viable if very large circuits need to be evaluated since it would consume large amounts of memory.

The garbling and evaluation are executed in parallel on different computing parties and times include the time spent on communication in addition to the computational time. Due to network layer instability issues, we were forced to make the garbler thread sleep for 40 ms after each batch of garbled tables, except

the last, was sent. 40 ms was enough so that the next batch would not be sent until the previous had been received by the evaluator. Therefore, the garbling and evaluation times of our prototype are roughly the same for larger circuits. For smaller circuits, the garbling time is almost always smaller, which is due to the communication overhead of sending garbled tables over the network, as evaluation is actually less computationally intensive than garbling.

For the oblivious transfer phase and the total runtime of the protocol, the mean was calculated from the maximum of reported times of all computing parties, meaning that only the reported time of the last computing party to finish was used in calculating the mean.

Circuit	**Input size (bits)**	**Total gates**	**non-XOR gates**	**Description**
Kreuter et al.'s circuits				
mil4	10	57	35	Solves the millionaire's problem for 4-bit values
mil128	258	1,793	1,027	Solves the millionaire's problem for 128-bit values
AES	384	50,935	16,070	Encrypts a 128-bit block with given key using AES-128 block cipher
edt-dist128	272	3,442,956	1,435,140	Calculates the edit distance of two 128-bit strings
dijkstra50	5,216	22,114,948	10,175,623	Computes the Dijkstra algorithm on a given 50-node graph
dijk-stra100	10,416	168,432,798	77,694,673	Computes the Dijkstra algorithm on a given 100-node graph
Tillich and Smart's circuits				
mult-32x32	64	6,995	5,926	Multiplies two 32-bit numbers
AES	256	31,924	6,800	Encrypts a 128-bit block with given key using AES-128 block cipher
SHA-256	512	132,854	90,825	Calculates the SHA-256 hash of a 512-bit block

Table 5: Descriptions of circuits used for benchmarking the hybrid protocol

Circuit	Initial parsing	Oblivious transfer	Garbling	Evaluation	Total
Kreuter et al.'s circuits					
mil4	0.1±0.7% 0.1±0.7%	37.4±1.5% 37.8±1.4%	0.17±0.6% 0.18±0.5%	0.03±0.3% 0.04±0.4%	45.6±0.1% 45.6±0.1%
mil128	0.2±0.7% 0.2±0.8%	40.0±1.5% 40.0±1.6%	2.3±0.1% 2.7±0.1%	3.2±9.2% 3.6±9.2%	51.0±1.1% 51.1±1.1%
AES	0.2±1.4% 0.2±1.5%	44.0±1.8% 44.8±1.7%	29.8±0.6% 34.4±0.6%	87.8±0.5% 92.4±0.5%	126.8±0.5% 132.9±0.6%
edt-dist128	0.2±2.8% 0.2±2.9%	41.0±3.3% 41.1±3.9%	3.88k±0.1% 4.29k±0.1%	4.00k±0.1% 4.40k±0.1%	4.04k±0.1% 4.44k±0.1%
dijkstra50	2.0±2.6% 1.9±3.2%	124.2±5.4% 124.8±5.5%	27.97k±0.2% 31.34k±0.1%	28.05k±0.2% 31.41k±0.1%	28.18k±0.2% 31.54k±0.2%
dijkstra100	5.3±1.6% 5.3±1.8%	311.3±4.6% 326.8±3.6%	226.8k±2.8% 252.3k±1.9%	226.8k±2.8% 252.3k±1.9%	227.2k±2.8% 252.7k±1.9%
Tillich and Smart's circuits					
mult-32x32	37.8±0.3% 38.3±0.3%	38.6±1.5% 39.2±1.6%	12.9±0.4% 16.3±0.5%	30.2±0.9% 36.0±1.3%	104.8±0.5% 111.3±0.6%
AES	101.4±0.2% 100.6±0.2%	43.1±1.6% 44.2±1.7%	16.1±0.8% 19.5±0.9%	38.3±1.3% 43.8±0.9%	182.5±0.4% 187.9±0.3%
SHA-256	768.8±0.3% 778.3±0.3%	92.8±5.1% 94.4±4.5%	195.8±0.3% 224.4±0.3%	287.3±0.3% 305.7±0.3%	1,146±0.4% 1,171±0.3%

Table 6: Performance results of the hybrid protocol in milliseconds with 95% confidence interval. For every circuit, the first row shows results with AES-NI instructions enabled and the second row with using OpenSSL v1.0.1 AES implementation

In [23], the authors of CBMC-GC present performance results of evaluating their compiled circuits using the secure two-party computation framework of [24], which also provides security in the passive model. They report the total time for garbling and evaluating a 32-bit multiplication circuit as 127 ms in a LAN experiment, whereas our experiment with the multiplication circuit took 30.2 ms for garbling and evaluation, even though the multiplication circuit we used contained about 3.5 times more non-XOR gates. The largest circuit benchmarked in [23] was the 8x8 matrix multiplication circuit, which contained 3.25 million gates with 900,000 non-XOR gates. The garbling and evaluation for that circuit took a total of 18.2 seconds. Our protocol evaluated a 128-bit edit distance circuit which is of comparable size, although slightly larger, in 4 seconds. Overall, our total evaluation times are several times smaller, which is probably due to the fast secret-sharing based oblivious transfer and the efficient garbling scheme with AES-NI by Bellare et al. [1] we used.

We can also compare our protocol's performance against protocols based on secret sharing. In [30], Laur et al. implement the AES-128 block cipher with secret-shared key and plaintext using only secret sharing, also on the Sharemind platform. They achieved a single AES encryption, including key expansion, in 652 ms. Compared to this result, our circuit-based approach achieves better performance for a single operation, namely using the AES circuit by Kreuter et al, we perform a single encryption in 127 ms, also including the key expansion. However, currently our protocol does no support parallelization in evaluating circuits and therefore, the cost for evaluating many circuits increases linearly. Laur et al. on the other hand are able to achieve an amortized cost of 0.37 ms for a single AES operation, since their protocol is easily parallelizable. Parallelizing the evaluation of many circuits seems to be a promising optimization in the future, since currently, running our protocol made use of only a fraction of the hardware capability and communication bandwidth available in our test environment.

Also, the garbling performance achieved by Bellare et al. [1] is significantly higher than our implementation, although the same garbling scheme is used. They report a garbling time of 637 μs for an AES-128 circuit compared to our 16 ms with Tillich and Smart's AES circuit. This performance difference can most probably be explained with Bellare et al.'s JustGarble system being highly optimized in the code level, using special compiler intrinsics to access SSE4 instructions and 128-bit registers to hold the wire tokens. Also, the performance gains they receive by using AES-NI instructions are much higher than we observed in our experiments. This suggests that our implementation could still be significantly optimized by non-cryptographic engineering efforts.

7 Conclusion

In this thesis, we have explored the seminal Yao's garbled circuits methods, which has enabled the construction of many SMC platforms and solutions. Already today, it is possible to develop applications for the secure manipulation of private data using SMC. However, the implementation of primitive protocols that can be composed to perform secure computations is often quite complex.

We have presented a general-purpose solution based on Yao's garbled circuits and secret sharing techniques, to simplify the construction of protocols that process secret-shared data. The hybrid protocol can be used in a variety of settings, where the number of parties associated with the secure computation application is arbitrary. The protocol's security is shown in the passive model against computationally bounded adversaries.

Two state-of-the-art Boolean circuit compilers from C were also analyzed and benchmarked. These compilers allow our protocol to be easily extended with arbitrary computational abilities. Together, the hybrid protocol and circuit compilers form an efficient toolchain for building robust SMC protocols.

We have described the prototype implementation of the hybrid protocol built into the Sharemind platform `additive3pp` protection domain. The prototype was tested and benchmarked with various different circuits. We also integrated the capability of evaluating circuits produced with the PCF circuit compiler into the built prototype.

The result of this thesis is an efficient general-purpose SMC protocol, that enables arbitrary computations on secret-shared data. We have improved on previous work in the area by using modern optimizations to the standard Yao's garbled circuits protocol and also making use of a refined garbling scheme. Our protocol also extends beyond the rather narrow two-party computational model. We have presented a proof sketch of our protocol's security in the passive model, using the Sharemind security proof framework. Our prototype succesfully demonstrates the capability of the hybrid protocol and achieves an increase in performance by orders of magnitude compared to the previous implementation. Our prototype's performance is also on par with results from standard implementations of Yao's technique, while still allowing for future improvements.

Using our protocol, constructing general-purpose efficient protocols for SMC is easy, as circuit compilers like PCF and CBMC-GC are able to produce the circuits needed to perform any calculation using our protocol. We have analyzed both compilers and seen that while CBMC-GC produces optimal circuits, PCF can compile fairly small and usable circuits in a fraction of the time it takes CBMC-GC to compile and optimize. Also, PCF enables evaluation of arbitrarily large circuits due to their memory-efficient circuit representation.

For future work, we found there is still much room for optimization in our

hybrid protocol prototype. Currently, we do not support evaluating many circuits in parallel, and as such, we are not using the available hardware potential to the fullest. Also, comparing our garbling benchmarks with the work of M. Bellare, we conclude that the prototype could be further optimized by non-cryptographic engineering efforts.

Since we have currently only considered security in the passive model, a natural step forward would be to extend our protocol to provide security against malicious adversaries. However, working in the malicious model adds a significant computational overhead and a careful study of available methods is needed to address this in an effective way.

Our future plan is to implement a complete suite of protocols using our hybrid protocol protoype by leveraging the capabilities of the circuit compilers. One potential candidate for this is a full secure protocol suite implementing floating-point arithmetic according to the IEEE 754 standard.

References

[1] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. *IACR Cryptology ePrint Archive*, 2013:426, 2013.

[2] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008.

[3] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In J. Simon, editor, *STOC*, pages 1–10. ACM, 1988.

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

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

[6] D. Bogdanov, P. Laud, S. Laur, and P. Pullonen. From input-private to universally composable secure multiparty computation primitives. In *CSF*. IEEE, 2014.

[7] D. Bogdanov, P. Laud, and J. 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.

[8] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[9] D. Bogdanov, R. Talviste, and J. 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.

[10] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In

R. Dingledine and P. Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[11] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[12] CBMC-GC. `http://http://forsyte.at/software/cbmc-gc/`, May 2014.

[13] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In J. Simon, editor, *STOC*, pages 11–19. ACM, 1988.

[14] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[15] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In S. Halevi and T. Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[16] I. Damgård and C. Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In T. Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576. Springer, 2010.

[17] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[18] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith. CBMC-GC: An ANSI C compiler for secure two-party computations. In A. Cohen, editor, *CC*, volume 8409 of *Lecture Notes in Computer Science*, pages 244–249. Springer, 2014.

[19] M. Geisler. *Cryptographic Protocols: Theory and Implementation*. PhD thesis, Aarhus University, 2010.

[20] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[21] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

[22] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security. CCS'10*, pages 451–462. ACM, 2010.

[23] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 772–783. ACM, 2012.

[24] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*. USENIX Association, 2011.

[25] L. Kamm and J. Willemson. Secure floating-point arithmetic and private satellite collision analysis. Cryptology ePrint Archive, Report 2013/850, 2013. `http://eprint.iacr.org/`.

[26] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[27] B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 321–336, Berkeley, CA, USA, 2013. USENIX Association.

[28] B. Kreuter and A. Shelat. Lessons learned with PCF: scaling secure computation. In M. Franz, A. Holzer, R. Majumdar, B. Parno, and H. Veith, editors, *PETShop@CCS*, pages 7–10. ACM, 2013.

[29] B. Kreuter, A. Shelat, and C.-H. Shen. Towards billion-gate secure computation with malicious adversaries. *IACR Cryptology ePrint Archive*, 2012:179, 2012.

[30] S. Laur, R. Talviste, and J. Willemson. From oblivious AES to efficient and secure database join in the multiparty setting. In *Applied Cryptography and Network Security*, volume 7954 of *LNCS*, pages 84–101. Springer, 2013.

[31] lcc, a retargetable compiler for ANSI C. `http://sites.google.com/site/lccretargetablecompiler/`, May 2014.

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

[33] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 715–724. ACM, 2011.

[34] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium. USENIX'04*, pages 287–302. USENIX, 2004.

[35] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[36] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. *IACR Cryptology ePrint Archive*, 2009:314, 2009.

[37] P. Pullonen. Actively secure two-party computation: Efficient Beaver triple generation. Master's thesis, Institute of Computer Science, University of Tartu, 2013.

[38] G. Seroussi. Table of low-weight binary irreducible polynomials. `http://www.hpl.hp.com/techreports/98/HPL-98-135.html`, 1998.

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

[40] The Sharemind secure computation platform. `http://sharemind.cyber.ee/`, May 2014.

[41] S. Siim and D. Bogdanov. A general mechanism for implementing secure operations on secret shared data. Technical Report T-4-21, Cybernetica, `http://research.cyber.ee/`., 2014.

[42] N. Smart and S. Tillich. Circuits of basic functions suitable for MPC and FHE, 2013.

[43] O. Šelajev. The use of circuit evaluation techniques for secure computation. Master's thesis, Institute of Computer Science, University of Tartu, 2011.

[44] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

Non-exclusive licence to reproduce thesis and make thesis public

I, Sander Siim (date of birth: 10th of August 1992),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1 reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2 make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Secure Multi-party Computation Protocols from a High-Level Programming Language

supervised by Dan Bogdanov and Sven Laur

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 14.05.2014