Practical evaluation of chain-like MPC protocols

Hanna Jonson
ha7008jo-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Paul Stankovski Wagner

Supervisor: Elena Pagnin

Examiner: Thomas Johansson

March 16, 2023

Abstract

Bottleneck Complexity (BNC) is a fairly new metric for Secure Multi Party Computation (SMPC) protocols. Two protocols with the BNC in mind are examined in this thesis. One protocol is successfully implemented in the Python programming language, and the BNC, latency and computational complexity of the protocol is examined with positive results. The BNC is independent on the number of parties in the protocol. The latency is linearly increasing with each new party added, and the computational complexity is linerarly dependent on the bitlength of the numbers computed on. The second protocol was not implemented because of the complexity of the Garbled Circuits that would have had to been created in order to perform encryption and decryption of an Additively Homomorphic Encryption (AHE) scheme. Instead, a conclusion is drawn that while garbled circuits serves a good purpose for secure computation in simple settings, computations that require complicated operations scale up very quickly in the number of gates needed. A github repository with the code for the first protocol is found at `https://github.com/ha7008/ChainMPC`.

Popular Science Summary

Secure Multi-party Computation (SMPC) allows computations where multiple parties participate, but only the result is revealed in the end. Imagine that you are a member of an exclusive board, voting for a new member to join. Only a unanimous decision allows a prospect to join, but it can be sensitive to share who have voted no. In the analogue case there is someone who counts the votes on pieces of paper. An electronic alternative is preferable in many situations.

Upgrading from pieces of paper to pieces of data could surely have been made in the reaping process of Suzanne Collins’ *The Hunger Games* as well. In choosing what child is to compete in the Hunger Games, a piece of paper is taken from a bowl with children’s names, some appearing many times as the poor families can trade food for their child’s name, increasing the likelihood that the child is chosen.

Both of these scenarios could have been solved electronically with the help of the two protocols suggested by Orlandi, Ravi and Scholl. The neat thing about these protocols is that they make the Bottleneck Complexity (BNC) independent on how many parties there is in the protocol. This means that even if a new party joins the protocol, the party that is already doing the most communication will not have to increase how many bits it sends or receives.

A protocol for the first usecase was implemented in the Python programming language. It makes use of a technique called Garbled Circuits (GC) that enables a party to perform computations without knowing what numbers are involved, only the result. The implementation showed that the BNC was in fact independent on the number of parties in the protocol. The second protocol, which could have been used in the Hunger Games, was not implemented because of the great complexity to create GC for this protocol. The GC would be used to perform encryption and decryption of an Additively Homomorphic Encryption (AHE) scheme. Encryption and decryption are two, usually very mathematically complex, algorithms that takes a message and turns it into a ciphertext, and then transforms it back into the original message. AHE is a very powerful technique when it comes to secure computing, allowing addition of the original numbers while keeping them secret as they are encrypted. Someone can do the computation without actually knowing the numbers, much like the GC itself.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

The field of cryptography is vast and to many people unknown. Perhaps the easiest entrance into the field is to think about the act of sending a message to someone. If Alice sends a message to Bob, through any popular messaging app or email service today, what guarantees that no one else online can read this message? This problem is normally tackled by having Alice encrypt the message before sending it, and Bob decrypting it when he receives it. If completely unfamiliar with this concept, a good read in Chapter 2 can provide the technical background. Encryption is good, but how does Bob know that the message actually came from Alice and not someone pretending to be Alice? Even if Bob knew that the message originally came from Alice, how does he know that no one tampered with it along the way? These are the kind of questions a cryptographer asks, and with each new question, the field of cryptography expands. Recently, a subfield within cryptography called Secure multi-party computation (SMPC) or sometimes just called multi-party computation (MPC) has grown alongside advances in cloud computing, mobile computing and distributed systems. The biggest difference now is that there is no longer only Alice and Bob, instead there could be thousands of parties participating, and they are not only sending messages to each other, they are performing computations that require inputs from many parties. SMPC is used for example in an electronic currency called Zcash [16] that has many similarities with Bitcoin.

Within this thesis, two multi-party computation protocols, for two different types of computations, are examined from a practical standpoint. One of the protocols was implemented in the Python programming language and one was deemed too complex to do it within a reasonable time frame. The primary purpose of the protocols, apart from performing the computations, is that the Bottle Neck Complexity (BNC) is independent on the number of parties in the protocol. This means that even if a party is added to the protocol, the party that is doing the most communication will not see an increase in communication. One of the main functionalities in these protocols is that no party will know what numbers they are computing on. Two cryptographic techniques used in order to achive this is Garbled Circuits (GC) and Additively Homomorhipc Encryption (AHE). Both these techniques enable a party to compute blindfolded in many other settings as well. Technical details are found in Chapter 2.

Through the following sections in this chapter one will learn about the concept of Secure multi-party computation, as well as the domain specific term bottle neck complexity. There is an overview to the two protocols and a usecase for each that aims to create an easy understanding for the reader about how the protocols can be used. Related work and the contribution of the thesis is presented. In the subsequent chapters, one is introduced to the cryptographic primitives and other priliminaries needed to understand the theory behind the protocols; the implementation considerations that have been made; an implementation explaination; the tests and how they have been performed along with the results; a conclusion; and last but not least, the discussion.

1.1 Secure multi-party computation

Secure multi-party computation (SMPC) or MPC enables multiple parties to jointy evaluate a function without any party knowing another party's input. Additionally, only the intended parties will know the answer of the computation. Originally introduced in 1982 by Andrew Chi-Chih Yao [41] his famous *Millionaires' problem* opened up the field for this research. In the Millionaires' problem there are two millionaires that jointy compute a function that determines who is richer, without revealing how much money any of the parties have. Later, this field has expanded to include many types of functions and problems, as well as more parties in the protocols. Typically, when there are only two parties it is denoted 2PC, and when there are three or more parties it is called MPC. A more technical definition of MPC is found in Section 2.3. Usage areas of MPC today include not only Zcash [16], but also encrypted databases, which do not only encrypt the data at rest but also through processing, distributed databases, key management, data analytics and more [4].

1.1.1 Bottleneck Complexity

Each party in an MPC protocol has a certain communication complexity, i.e. how many bits it has to send and receive and how it relates to for example how many other parties that are in the protocol. The party with the largest communication complexity will be called the bottleneck, and the Bottleneck Complexity (BNC) for the protocol is equal to the maximum communication complexity [7] of any party. Minimizing the BNC of a protocol can be of great interest if the protocol is meant to run on lightweight devices, as for example mobile phones, IoT-devices or other restricted machines. Technical definitions of BNC is found in Section 2.3.4.

1.2 Chain-like MPC protocols of Orlandi, Ravi and Scholl

In the paper [34] the authors propose two MPC-protocols with BNC that is independent on the number of parties taking part in the protocol. This means that even if more parties join the protocol, the BNC does not increase. Both protocols includes a setup phase and three active phases. All parties in the protocols are ordered in a chain-like manner, meaning that each party does only communicate with the previous and the next party in the chain.

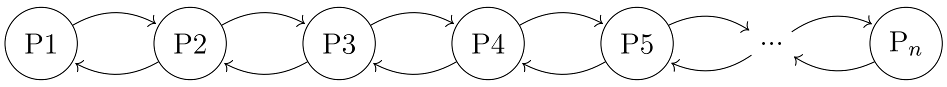

Figure 1.1: Parties in the protocols. Each party communicates only with the next and the previous party. This is why it is called a "chain".

1.2.1 Introduction to Protocol 1

The first protocol regards abelian programs, essentially computing a function on the sum of all the parties inputs. In the implementation, the sum is calculated and then the function is to check whether the sum is the same as a value V. Each party's value can be denoted v_i for each index i. This can be described mathematically as:

$$\text{result} = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} v_i = V \\ 0 & \text{else} \end{cases}$$

This is the result of the protocol, and will be shared with each participant. How to get there is not exactly straight forward, but the process is described below.

Setup phase of Protocol 1

In the setup phase, a Garbled Circuit (GC) is constructed that is used to perform the actual computation. It is a sort of boolean circuit, and this is given to the first party P1 and enables it to do the computation without knowing what values are present in the computation, only the result in the end. Exactly how a GC works is explained in Section 2.4. However, one needs to know that there will be a number of labels, essentially random keys, used as inputs to the GC. One label per bit in the number that is worked with is needed as input. However, as each bit can be either a 1 or 0, there will be two different labels per bit but only one will be used. Each label is split into n additive shares, one for each party, meaning that the sum of all shares constructs the label. Each party gets a share for every label. Each party will also receive a randomness, a random number, which will be seen used already during the first forward pass of the chain.

Phase 1 of Protocol 1

During the first forward pass in the chain, each party will send to the next the value that they choose. They will add this to whatever they have received previously, meaning that in the end, the last party will have received the sum of all parties' inputs. This is good but not enough. For example, the second party would know for certain what the input of the first party was. Each party therefore also adds a random number r_i to their value, masking it and making it impossible to know what their value is. This is the randomness that they received during the setup phase.

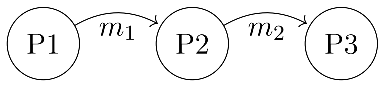

Figure 1.2: The first forward pass in the chain. Note that $m_1 = v_1 + r_1$ and $m_2 = v_1 + v_2 + r_1 + r_2$

In the end, the last party will have received the sum of all parties inputs plus the sum of all randomnesses, this is called Z.

$$Z = \sum_{i=1}^{n} v_i + r_i$$

Phase 2 of Protocol 1

In the second phase a backward pass through the chain is performed. Each party has two label shares for each bit in Z. For each bit, if it is 0 the party will send the corresponding label share, but if the bit is 1 the party will send the other one. Each message will therefore be constructed of Z and the sum of all label shares that has passed the chain so far.

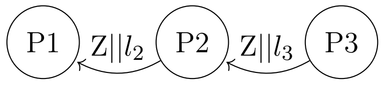

Figure 1.3: Backward pass in the chain. Z and the label shares are sent backward. Note that l_3 is the label shares from only P3, and l_2 is the label shares of P2 + l_3.

In the end of the backward pass, P1 will have the sum of all label shares for each bit in Z. This means that P1 has all the complete labels. P1 will then use these labels as input to the Garbled Circuit which it received during the setup. The GC has the value R hard coded into it, where R is the sum of all randomnesses. The GC performs a computation $f(Z - R)$. Mathematically, one can see that:

$$Z = \sum_{i=1}^{n} v_i + r_i$$

As concluded before, Z is the sum of all parties' values plus their randomnesses.

$$R = \sum_{i=1}^{n} r_i$$

R is the sum of all randomnesses.

$$Z - R = \sum_{i=1}^{n} v_i + r_i - \sum_{i=1}^{n} r_i = \sum_{i=1}^{n} v_i$$

$Z - R$ is the sum of all parties values.

The function $f(Z - R)$ therefore is a function on the sum of the parties input values. In the implementation, this is to check whether this is a number V or not. After having evaluated the GC, P1 has received the result from the function, which can be either a 0 or a 1.

Phase 3 of Protocol 1

During the last phase of the protocol, the result is distributed through the chain with a forward pass.

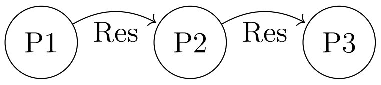

Figure 1.4: Distribution of the result.

1.2.2 Use case for Protocol 1

One may ask where protocol 1 could be used in real life. A typical scenario for an abelian program regards voting. Specifically let's look at a unanimous decision regarding electing someone to an exclusive board. Each party can only vote yes (1) or no (0). In this case the number V is equal to the number of parties n in the protocol. The function that is computed can then be defined as:

$$\text{result} = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} v_i = n \\ 0 & \text{else} \end{cases}$$

1.2.3 Introduction to Protocol 2

The second protocol regards a selection function. The first party will select an index, and through the protocol learn the value of the party with that index. The first party's input is the selection index where $q \in \{2, ..., n\}$. Mathematically the function is described:

$$f(x_1 = q, x_2, x_3, ..., x_n) = x_q$$

The interesting thing is that no other party will learn what the selection index was, and no one will know if their value was chosen or not. In order to achieve this, the protocol makes use of both Garbled Circuits (GC) and Additively Homomorphic Encryption (AHE). Intuitively, AHE allows one to perform addition with the numbers that are encrypted. The process of the protocol is described below.

Setup phase of Protocol 2

In the setup phase, each party except the first one will receive a Garbled Circuit that can perform encryption using an encryption scheme that has additively homomorphic properties. It will check whether two values are the same, and if they are it encrypts a 1, otherwise it encrypts a 0. The first party is given a GC that performs decryption using the same scheme. The parties are also given label shares and labels that can be used as input to the GCs.

Phase 1 of Protocol 2

The protocol begins with a forward pass through the chain, just like protocol 1. The first party selects an index, and chooses the labels that are corresponding to the bits in that number. These labels are then passed to the second party who uses the received labels along with their own labels as inputs to the Garbled Circuit for encryption. If the two values match, the party will receive an encryption of a 1, otherwise an encryption of a 0, but the party will not know which one was received. The party then performs scalar multiplication with the ciphertext and their value. This means that if the party’s value was 3 and the encrypted number was 0, the result will be an encryption of $3 \cdot 0 = 0$, but if the encrypted number was 1, the result will be an encryption of $3 \cdot 1 = 3$.

The second party then sends their ciphertext, which is the encrypted number, to the next party, along with the labels received from the first party that corresponds to the selection index. The third party does the same thing as the second party, but before sending the ciphertext, uses the additively homomorphic properties of the encryptions to ”add” them. If the third party received an encryption of a 3 from the second party, naturally it will create an encryption of 0. ”Adding” them together creates an encryption of $3 + 0 = 3$. The exact mathematical operations to do this depends on the encryption scheme. The same patterns follow along through the chain, meaning that in the end the last party will have an encryption of $3+0+0+0+0+0... = 3$ if the selection index in this case was 2 and the second party’s value was 3. No party will of course know that this is the case, only that there exists a ciphertext, which is an encrypted number, at the last party that holds the chosen value.

Phase 2 of Protocol 2

The second phase of Protocol 2 is for logical purposes identical to the one of Protocol 1. Each message will be constructed of the ciphertext and the sum of all label shares that has passed the chain so far. In the end, the first party has the sum of all label shares for each bit in the ciphertext and this creates the final labels used as input to the Garbled Circuit for decryption. The first party evaluates the GC and receives the decrypted value it selected, and in the example we have followed so far that value will be 3.

Phase 3 of protocol 2

The last phase of the protocol is identical to the one of the first protocol. The result is distributed through a forward pass in the chain.

1.2.4 Use case for Protocol 2

To find a good use case for the second protocol one will have to think about a special feature of the second protocol - multiple parties can have the same value. This means that if more parties join with the same value, the chances of that value getting chosen increases, if the first party selects randomly. The avid reader might recall that this resembles exactly the reaping process in Suzanne Collins’ *The Hunger Games* [12]. While the book version makes use of of pieces of paper with names written on them, the new improved version is computerised for an even more futuristic feel.

1.3 Related work

The closest related work to this thesis is, of course, the work of Orlandi, Ravi and Scholl [34] upon which the thesis is built, which is discussed throughout the thesis. Relating to implementation work of MPC there is a project from Boston University in collaboration with other organisations [28] creating open source libraries for MPC [29]. Regarding protocol communication patterns, there is [21] showing promising results for a chain-like pattern in relation to communication complexity. For the reader interested in Garbled Circuits, [10] provides an overview of recent work.

1.4 Subject of study

The purpose of this thesis is to examine the protocols from a practical point of view. Three areas are investigated, the BNC, latency and computational complexity. These are all metrics that affect the usability of the protocols in real life. Additionally, effort have been put into seeing where the protocols lack in enough efficiency for an implementation during the time frame of the thesis work.

1.5 Methodology

The methodology includes creating a program that run simulations upon which the BNC and latency are measured. The experimental results include data points that shows practical results, in contrast to estimating a theoretical result. The gain in getting data points from practical results instead of theoretical estimations is that it closer represents usage in real life. The program is only run on a single machine as it eliminates the need for network communication. The computational complexity is not measured, but estimated through calculating the mathematical operations that are computed. From the results conclusions are drawn about the three areas, BNC, latency and computational complexity. Additional insights are given about the practicality of the protocols after the effort of implementing the first protocol and seeing where the second protocol lacks in efficiency from an implementation perspective.

1.6 Contributions

Through this thesis three areas of contribution are achieved.

- Protocol 1 is successfully implemented as a proof of concept for this idea. The BNC along with latency and computational complexity was examined.
- Protocol 2 was not implemented, but an insightful conclusion is drawn regarding the use of Garbled Circuits for complex operations.
- A github repository containing the code for examination can be found at `https://github.com/ha7008/ChainMPC`

Contribution regarding implementation details that had to be created by the author rather than found in open source are included below.

- Change of data structure for the GC from a binary tree to an array.
- Selection of labels enabled at any level in the circuit.
- Splitting labels into shares and putting them back together.
- Reusability of labels.

For details about this implementation contribution, see Section 3.5.2.

Chapter 2 Cryptographic primitives and other preliminaries

This chapter introduces the cryptographic primitives that are used in the two protocols by Orlandi, Ravi and Scholl [34], along with notation and other definitions needed to be able to read and understand the thesis painlessly. The primitives are explained individually for any reader that is new to the field of cryptography, or haven't learned about each of the primitives yet. After all primitives have been introduced, their place in the two MPC-protocols will be specified and explained. In the end of this chapter, latency and computational complexity will be explained as these are the two metrics that will be the foundation of the evaluation of the protocols.

2.1 Encryption schemes

Confidential communication can be achieved through encryption of a message, often called a plaintext, and turns it into a socalled ciphertext. The ciphertext should appear random, so that no information about the plaintext is revealed by examining the ciphertext. Together with the plaintext, a key is taken as input to the encryption algorithm. While having the ciphertext and, depending on the encryption scheme, the same key or another key, one can use decryption to turn it back into the correct plaintext.

From [23] one can learn the formal notation for an encryption scheme. The ciphertext c can be obtained through running the encryption algorithm Enc using the key k for the plaintext message m. Formally it is expressed

$$c = Enc_k(m)$$

Decryption is denoted in a similar fashion

$$m = Dec_k(c)$$

Another simpler notation sometimes used when the key is implicit is

$$c = E(m)$$

$$m = D(c)$$

2.2 Additively Homomorphic Encryption

Through [24] one can learn about partially homomorphic encryption schemes. Fully homomorphic encryption entails that there exists an operation on two ciphertexts that will correspond to the addition of their plaintexts, as well as another operation that correspond to multiplication. Partially homomorphism means that not both addition and multiplication are boundlessly possible. In the work [34] it is stated clearly that only additively homomorphic encryption is needed for the second protocol, enabling us to focus only on this aspect. Homomorphic encryption (of any sort) ultimately enables one to do computations on encrypted data.

Mathematically, additively homomorphic encryption can be expressed as:

$$D(E(m_1 + m_2)) = D(E(m_1) \boxplus E(m_2))$$

where $\boxplus$ denotes the operation that can be computed on ciphertexts and returns an output ciphertext that corresponds to addition of plaintexts. Note that $\boxplus$ need not be addition itself, however decrypting the result of the operation on the two ciphertexts will produce the same result as if one had just added the two plaintexts before encryption. Examples of such encryption schemes include ElGamal [17] and Paillier [35].

Additionally, fully homomorphic encryption is not considered, as it is stated by [34] that the protocols are specifically developed under assumption that FHE is avoided.

2.3 Secure multi-party computation

A short introduction to MPC was given in section 1.1, and therefore this section will focus more on the technical details and concepts. Through the "IEEE Recommended Practice for Secure Multi-Party Computation" [1] one can learn about the concepts for MPC in a comprehensive manner while being relevant to where this technology is at present.

2.3.1 Adversary model

The adversary is an entity in control of one or more parties taking part in the MPC-protocol. A malicous adversary may actively try to disrupt the computation or take any action it wants, a covert adversary can take mallicous actions but will do so trying not to "get caught", and a semi-honest adversary will follow the protocol specification while trying to learn as much as possible from the output [1]. According to [34] the protocols suggested are only safe against semi-honest adversaries. In other words, an adversary taking another action than what is specified in the protocols, is not taken into account in the protocol design. For a more nuanced adversary model, including the adversary’s ability to recruit and abandon corrupt parties as well as differences in computational power, see [9].

2.3.2 Security requirements

According to [1] there are two fundamental requirements and three optional requirements for an MPC-protocol. The two fundamental requirements are

- **Input Privacy**
  No party can learn any information beyond the computational result.
- **Result Correctness**
  If the protocol outputs a result, it will be lossless compared to the same protocol run in a plaintext-manner.

The tree optional requirements are

- **Fairness**
  If an adversary receives the result, so will the honest parties.
- **Guaranteed output delivery**
  An adversary can not prevent honest parties from receiving the result.
- **Probability to catch deviation**
  If a party behaves dishonestly, there is a probability to catch the corrupt party.

2.3.3 Communication patterns and roles

There are many different communication patterns a protocol can have. The chain-like communication pattern seen in figure 1.1 is one of them. Other patterns include, but are not limited to, unrestricted communication, star, and directed acyclic graph.

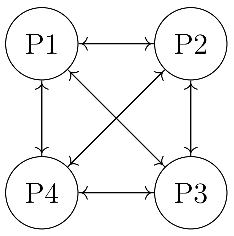

Figure 2.1: Unrestricted communication, all parties can communicate with each other.

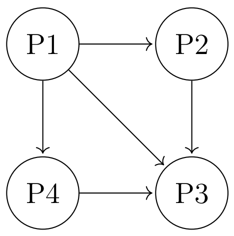

Figure 2.2: Directed acyclic graph. There are no cycles in the communication pattern.

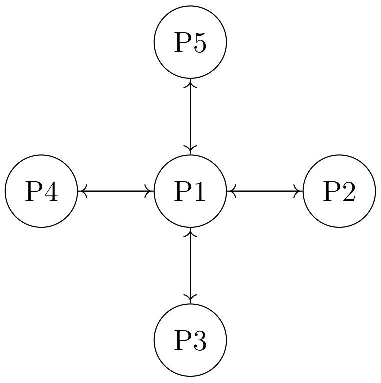

Figure 2.3: Star communication pattern, all parties communicate with a node in the middle, creating an appearance of a "Star".

A chain-like communication pattern is favourable for a low communication complexity of an MPC-protocol. For further details look into [21].

Considered in [1], different parties have different roles in an MPC-protocol in all real life applications. Roles mentioned are *Task initiator* who is the first party that will go to the coordinator to initiate the task; *Data provider* who is a party in the protocol that will provide data to the computation; *Algorithm provider* who is a party that will provide the algorithm that will be run; *Coordinator* who will coordinate all the other MPC participants; *Computing provider* who is a party that will be doing the computation; and *Result obtainer* who will be a party that receives the result. Some of the roles may overlap, however it is recommended that only one participant takes the role of coordinator as well as only one participant taking the role of algorithm provider. For the protocols of this thesis all participants have fairly equal roles, all being *Data providers*, *Computing providers* and *Result obtainers*. Who the coordinator is will be an open question for the discussion in Chapter 7.

2.3.4 Bottleneck complexity

Bottleneck complexity (BNC) was first introduced in the work [7], and is a metric for the maximum communication complexity of any party in the MPC protocol. This regards how many bits any party will have to send and receive, and how that number grows in relation to the number of parties in the protocol or other affecting variables. Relating to the usecases in this thesis, an affecting variable for the BNC will naturally be the message size.

Denote the number of parties in the protocol n. If one party communicates with all other parties, sending or receiving one bit, the BNC would naturally be $O(n)$ - the BNC is linearly dependent on the number of parties in the protocol. Instead, in the chain like protocols of Orlandi, Ravi and Scholl [34], the BNC is independent on the number of parties of the protocol.

2.3.5 Correlated Randomness

Through [34] we learn that correlated randomness is used in the protocols. The basic idea is to take a set of random numbers, that are uncorrelated, and determine a relation between them, which creates a correlation. In this case, it is the sum of the random numbers that will make the correlation. In the general case it could be multiplication or another relation. No random number in itself will reveal anything about the others as they were independently chosen from an agreed upon distribution.

For the protocols in this thesis, the relation of the correlated randomness is mathematically expressed as:

$$R = \sum_{i=1}^{n} r_i$$

where n denotes the number of parties in the protocols in this case, and r_i is representing each of the random numbers.

2.4 Garbled Circuits

The introduction of Garbled Circuits [2] is typically credited to Andrew Chi-Chih Yao and his paper "How to Generate and Exchange Secrets" [42] from 1986. Although there is no mention of a "garbled circuit", the paper introduces a new tool for two parties to be able to compute any function in the form of a boolean circuit while remaining privacy of the parties inputs throughout the computation. A simpler explaination of the garbled circuit technique can be found in [10] with information about current improvements. The basic idea is as follows:

Imagine two parties, Alice and Bob, who want to compute an AND-function of their private inputs. Alice will be the one constructing the garbled circuit and Bob will be the one evaluating it.

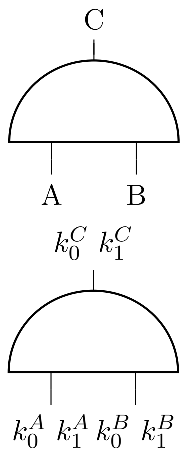

Figure 2.4: At the top: a normal AND-gate. At the bottom: an AND-gate with respecitve labels.

A	B	C	Encrypted Value
0	0	0	Unencrypted
1	0	0	Unencrypted
0	1	0	Unencrypted
1	1	1	Unencrypted

Table 2.1: Truth table for regular AND-gate.

A	B	C	Encrypted Value
k_0^A	k_0^B	k_0^C	$E_{k_0^A}(E_{k_0^B}(k_0^C))$
k_1^A	k_0^B	k_0^C	$E_{k_1^A}(E_{k_0^B}(k_0^C))$
k_0^A	k_1^B	k_0^C	$E_{k_0^A}(E_{k_1^B}(k_0^C))$
k_1^A	k_1^B	k_1^C	$E_{k_1^A}(E_{k_1^B}(k_1^C))$

Table 2.2: Truth table for garbled AND-gate.

Illustrated in Figure 2.4 are two boolean circuits computing an AND-function. The top one represents a "normal" AND-gate, where the input can be either 0 or 1 on both A and B. If both A and B are 1, the output C will be 1 as well, otherwise 0. The bottom circuit represents a garbled circuit where, instead of 0 or 1, all inputs and outputs will be replaced with a randomly sampled key called a label. Both input labels (one for A and one for B) will be used to encrypt the corresponding output label of the gate, in a socalled double encryption. Note that the encryption scheme is symmetric, meaning that the same key is used both for encryption and decryption. The corresponding truth tables can be seen in table 2.1 for the normal AND-gate and table 2.2 for the garbled AND-gate.

Alice has constructed the garbled truth table and will send it to Bob. However, in order to preserve privacy of Alice's input, the encrypted values will be permuted, the order of them will be changed, before Alice sends the table. Also note that Alice will not send the input values, only the encrypted output values, after having permuted them.

A	B	C	Encrypted Value
k_0^A	k_0^B	k_0^C	$E_{k_0^A}(E_{k_0^B}(k_0^C))$
k_1^A	k_0^B	k_0^C	$E_{k_1^A}(E_{k_0^B}(k_0^C))$
k_0^A	k_1^B	k_0^C	$E_{k_0^A}(E_{k_1^B}(k_0^C))$
k_1^A	k_1^B	k_1^C	$E_{k_1^A}(E_{k_1^B}(k_1^C))$

$\rightarrow$

Encrypted Value
$E_{k_0^A}(E_{k_1^B}(k_0^C))$
$E_{k_1^A}(E_{k_1^B}(k_1^C))$
$E_{k_0^A}(E_{k_0^B}(k_0^C))$
$E_{k_1^A}(E_{k_0^B}(k_0^C))$

Table 2.3: Permutation of truth table with garbled values.

Alice will also send the key labeled with her secret input value to Bob, i.e k_0^A or k_1^A, but to Bob this will only look like a random value (which it technically is so far). Additionally, Bob will need his key, k_0^B or k_1^B in order to decrypt the correct value in the table. He needs to get it from Alice, but if he tells Alice which key he wants his privacy will not be preserved. Therefore they will run a 1-out-of-2 Oblivious Transfer (OT) protocol in order for Bob to get his key without Alice knowing which one he chooses. In this way, Bob will only get the one key he chose, and not both. If he had gotten both he could have derived information about what Alice's input was as he could correctly decrypt two of the ciphertexts.

In this example, Bob will of course know that Alice chose 1 if he also chose 1 and the output was 1. It is easy to derive information from a circuit this simple. However, more complex circuits can be built in a layered approach, having great potential for more complex MPC-protocols.

In the case of the protocols [34] of this thesis, no OT is needed as the choice of labels isn't secret. Instead, in order to prevent the evaluator of the circuit from being able to evaluate the circuit many times with different results, only one label out of the two possible per bit can be constructed from the label shares.

2.5 Abelian programs

Abelian programs are well explained in [34] as ”functions on the sum of the parties' inputs over an abelian group”. The end result should become either false or true, a 0 or a 1. Mathematically this is notated

$$f : G \rightarrow \{0, 1\}$$

2.6 Selection functions

A selection function is essentially taking as input an index and a set of values, and outputs the value that is at the index selected. Mathematically it follows the notation $f(x_1 = q, x_2, x_3 ... x_n) = x_q$ through the paper [34]. Naturally, the index can not be 1 as that is the index of the selecting variable, and the maximum allowed value of q is n.

2.7 Latency

Latency describes how much time it takes for information to go from one place to another. Low latency is often considered better than greater latency. In the case of protocols like these, the total latency can be described as the time it takes to run the protocol from start to finish, from when the first packet of data is sent until the last one is received.

2.8 Computational complexity

Computational complexity describes how resource usage increases depending on a variable. Usually it is denoted $O(n)$ for a linear relationship. However, what is more important than the exact coefficient, is what type of relationship is prevalent. For example one is usually looking for if the complexity is $O(n)$, $O(n^2)$, $O(log_n)$ or in some cases $O(1)$ where the 1 denotes that there is no relationship between the computation and the variable, usually because the computation is at a stable level regardless of the variable fluctuation.

Chapter 3

Implementation considerations

There are many things to consider for the implementation process. One of the most impactful choices is what programming language to use, while which libraries to use for Additively Homomorphic Encryption (AHE) and Garbled Circuits (GC) are also significant. Below is described things that are good and bad with different choices, and in the end the finalized selection is presented.

3.1 Programming Language

The most important thing to consider when choosing a programming language regards the availability of libraries needed for the implementation. Only languages that had any available libraries for AHE and/or GC open source were considered, however, some language specific features are considered as well.

Considered programming languages			
Language	Performance	Virtual compartmentalisation	Ease of use and understanding
Python	Slow	No	Very Easy
Java	Medium	Yes	Easy
C++	Fast	No	Hard
Go	Fast	No	Easy
JavaScript	Fast	No	Easy

Table 3.1: Considered programming languages.

The considered programming languages were, as can be seen in table 3.1, Python, Java, C++, Go and JavaScript. Cryptography-wise it is usually desirable that the language is ”fast”, practically meaning that an implementation of an algorithm will run fast enough for us to be able to be able to produce simulation results. This is because the numbers are usually very large and depending on how certain mathematical operations are implemented in that language, it can take a lot of time to run the algorithm, or not.

Another desirable trait is that the language can compartmentalise so that different parts of the code can not reach one another. This is the case with for example Java where a virtual machine makes sure that attributes declared as private are actually private. In contrast, with for example Python nothing can be seen as private in that sense. If a function or variable has been declared it can be reached. This can be of importance when the whole system can not be trusted, however the goal of this thesis is not to produce a fully secure implementation.

More importantly than the performance or the compartmentalisation is however the ease of implementation for this thesis. In this aspect there is one language that outperforms the other candidates greatly. Python is not only known as a very beginner friendly language, but it also requires very little set up and has a very easy installation process for new libraries. Additionally, while it is in many aspects slower than the other candidates, it performs fast enough for simulations to be made.

3.2 Additively Homomorphic Encryption algorithm

The chosen Additively Homomorphic Encryption algorithm mentioned in the paper [34] needs to satisfy the following conditions.

- Should be a public key encryption algorithm, in other words asymmetric.
- Needs to support additive homomorphism.
    - Have a function for addition of two ciphertexts, ex $c_1 + c_2$.
    - Have a function for scalar multiplication, ex $c_1 \times 3$ (which is the same as $c_1 + c_1 + c_1$).
- Works over the ring of integers modulo M, $(\mathbb{Z}_M, +)$.
  Note: this is an assumption rather than a requirement as it is stated in the paper [34] that it would be possible to use small integer plaintexts rather than $\mathbb{Z}_M$.

Through [24] one can learn about a number of partially homomorphic encryption algorithms, which are listed below together with their respective fit to the requirements.

Partly Homomorphic Encryption Algorithms from [24]					
Algorithm	Asymmetric	Addition of ciphertexts	Scalar multiplication	$(\mathbb{Z}_M, +)$	Comments
RSA	Yes	No	Yes	Yes	Widely available
Goldwasser-Micali (GM)	Yes	Yes	Yes	Yes	Large overhead
ElGamal	Yes	Yes	Yes	Yes	Additively homomorphic if modified
Benaloh	Yes	Yes	Yes	Yes	Generalisation of GM
Naccache-Stern (NS)	Yes	Yes	Yes	Yes	Generalisation of Benaloh
Okamoto-Uchiyama	Yes	Yes	Yes	Yes	
Paillier	Yes	Yes	Yes	Yes	
Damgård-Jurik	Yes	Yes	Yes	Yes	Generalisation of Paillier
Boneh-Goh-Nissim (BGN)	Yes	Yes	Yes	Enough	See below for explaination
Sander-Young-Yung	Yes	No	No	No	

Table 3.2: Partly Homomorphic Encryption Algorithms from [24] and their fit to the requitements of the AHE algorithm for the protocol for selection functions of [34].

From the table above one can conclude that among the examined algorithms, there are 8 of which could possibly fit into the protocol. Each of these will briefly be discussed below. An emphasis is placed on the ratio between the lengths of the plaintext and the ciphertext (sometimes called expansion rate or expansion factor). A ratio of 1 means that the ciphertext is as long as the plaintext, while a ratio of 2 means that the ciphertext is twice as long as the plaintext. A smaller ratio is therefore desireable as the purpose of the protocols subject in this thesis ([34]) is detatching the BNC from the number of parties of the protocol. That should reasonably not also imply an increase of any communication complexity, which would be the case if the ciphertext has a greater length than necessary. Only implementations enabling use of the homomorphic properties of the algorithms are considered, in other words there have to exist functions or methods for addition of ciphertexts as well as scalar multiplication that are readily available and usable.

- **GM**
  Proposed in 1982 [20], the Goldwasser-Micali cryptosystem offers a probablistic encryption scheme. However, as noted by [24], the ciphertext is many times longer than the plaintext, making it an inefficient algorithm in our application.
  GM is implemented in Java [3].
- **ElGamal**
  In 1985 the ElGamal cryptosystem was introduced [17], which is also a probabalistic encryption scheme, with the ciphertext expected to be twice the length of the plaintext. Originally, ElGamal is a multiplicatively homomorphic encryption scheme, however, with slight modification it can become additively homomorphic [24], and therefore fit the requirements.
  ElGamal is implemented in Java [3].
- **Benaloh**
  The Benaloh encryption scheme [11] introduced in 1994 is an improvement on the GM scheme where efficiency is drastically improved. Unlike GM, Benaloh can encrypt more than one bit at once using the same length randomness.
- **NS**
  In 1998 the NS cryptosystem was introduced [31], which offers one deterministic and one probabilistic version. According to [24] NS can be viewed as a generalisation of the GM and Benaloh algorithms while improving the expansion rate, in other words the ratio between the plaintext and ciphertext, making it more efficient than both the GM and Benaloh algorithms.
- **Okamoto-Uchiyama**
  In 1998, another probabalistic encryption scheme was introduced by Okamoto and Uchiyama [33]. The size of the ciphertext is three times the size of the plaintext.
- **Paillier**
  Pascal Paillier introduced a probabalistic encryption scheme in 1999 [35], with a ciphertext size twice the plaintext size.
  Paillier is implemented in Python [15] [25], Java [3], Go [38], C++ [43].
- **Damgård-Jurik**
  In 2001 Damgård and Jurik proposed a generalisation of the Paillier cryptosystem [14], where the expansion factor is decreased from 2 to almost 1.
  Damgård-Jurik is implemented in Go [26], Python [13], C++ [22].
- **BGN**
  In 2005 the BGN algorithm was proposed [6], and enables additive homomorphism for a smaller plaintext space than $\mathbb{Z}_M$, as the decryption relies on computing the discrete logarithm. However, it is supposedly big enough for the intended application of this thesis.
  The BGN algorithm is implemented in C++ [36].

3.3 Garbled circuit scheme

A special requirement for the garbled circuit scheme for the protocol regarding selection functions that is mentioned in the article [34], is that it should be able to take as a input the labels of the garbled circuit. Another consideration is that the garbling scheme should be usable for multiple parties, in contrast to only 2PC. This limits the number of libraries available as many have been implemented for 2PC only. In these cases the functions needed for garbling and ungarbling the circuit were located logically within a pre-defined party. In order for this to work well in a more complex setting, the circuit has to be abstracted in a way in which it can be used without that pre-defined party. This is because the parties in the protocols of [34] will have other functionality as well, meaning that they cannot work well in the narrow context that is provided in many naive open source libraries.

Current libraries available are in C++ [18] [39] [19] [37], Java [8] [40], JavaScript [30], Python [32], Go [5]

3.4 Testing

A testing framework is used in order to evaluate the latency of the protocols. Simple time measurements are enough for basic tests and functionality exists in most languages. A more fine grained framework for multiple purposes has been found available in Java [27], though other frameworks seem to test latency of specific applications.

3.5 Finalized selection

The finalized selection of programming languages and libraries consists of

- Programming language: Python 3.8.1
- Garbled circuit library: gabes [32]
- Additively homomorphic encryption scheme library: never chosen

3.5.1 Python

The main advantage of Python, compared to for example Java or C++ who were other strong candidates in the selection process, is that there is very little preparation, configuration or installation needed to run programs with Python. It is very simple to run something, to run anything, and to run everything. It is a fairly simple programming language that fits many beginner programmers, meaning that most academics within technical fields will be able to examine the code fairly easily, while also being able to test and run it themselves without too much effort.

Libraries for both Java and C++ were failed to install and use easily. For Java the problem was that the two libraries chosen for AHE and GC were using different Java versions. For C++ the problem was that it is was not fit for the specific architecture of the intended machine.

Another great benefit of the Python programming language is that it is very easy to extend or alter someone else's code - as has been necessary in this project. With for example Java programs one is usually given binares that can be run, but hardly altered.

Considering the pros and cons of the Python programming language it was the best fit for this project. The project is neither about creating a perfectly secure solution, nor creating the best (fastest) solution, it is about creating something that works, that can be cross-checked by other academics and that proves a concept, all within a very limited time frame.

3.5.2 gabes

Checking the different libraries for garbled circuits, there was nothing that was "perfect", meaning that not all the functionality desired was provided. This includes the ability to create arbitrary circuits easily, to create shares of labels, to program predefined values into the circuit or handle label selection in a well fashioned manner.

gabes [32] provides easily extendable classes with many selections for the garbling and ungarbling mechanisms. Originally, the mechanism that constructs the circuit takes as input a file of instructions that need to correspond to a sort of balanced binary tree. This is practical for simple circuits and demonstration purposes, such as for example using a small circuit with a few AND-gates or such, but it is not very good for more complex circuits such as a regular subtraction circuit. Depending on the size, it includes hundreds of gates that have to be ordered in specific ways. Also, in a subtraction circuit, the inputs and outputs of a gate has to be usable many times, which is not the case if the circuit has to be built like a balanced binary tree. Some other libraries can provide other data structures for the circuit, but lacks in usability with limitations such as only allowing three layers in the circuit. Hence, there was no perfect library, but gabes proved to be easily extendable and provide much of the base functionallity needed.

Extensions that were made to the library were

- The binary tree structure of the circuit was omitted, and an array-like structure with an arbitrary number of layers was implemented. Each subpart of the circuit could be defined on its own and reused. A specific circuit needed to be predefined and could not be built on the go, however, technically any circuit could be constructed with limited effort. The order of garbling and ungarbling of the gates could be different for each circuit part, meaning that there was no uniform way of doing it, unlike the binary tree structure where it follows the leaf-to-root direction.

- Label selection could be at an arbitrary level, and provided as a built in function or taken as a parameter. Originally, only labels that were at the leaf levels could be chosen, and no predefined values could be used.
- Labels could be split into many shares, and the shares could be put back together into a usable label.
- Labels (and wires) could be reused for more complex circuits where inputs and outputs should be used more than once.

Chapter 4 Implementation Explanation

For the interested reader, this chapter will aim to provide a closer explanation to how the code works and what it does.

4.1 Protocol 1

First introduced in Section 1.2.1, the first protocol computes the function $f(Z-R)$, where f denotes a function comparing the input to a fixed value V. Let there be n participants. In this explanation the case is when $n = 3$. The first party is the Evaluator party and the two other parties are normal parties. In addition, there is also an entity that is the Algorithm provider. This could potentially be one of the parties, however for simplicity it's *not* one of the parties. More on this topic will be covered in the discussion.

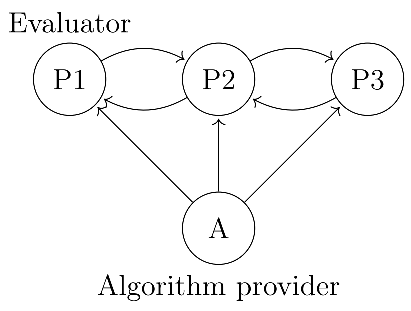

Figure 4.1: Parties in the protocol.

Each party can communicate with the previous and the next party in the chain. Notice that the first and last parties are not communicating with each other. The Algorithm provider can, however, send information to all parties during the setup phase. This chain could be extended to include an arbitrary number of parties. The first party is always the Evaluator party.

4.1.1 Setup phase

During the setup phase there are a few things that are happening.

- The Algorithm provider creates the garbled circuit that computes $f(Z-R)$, and sends it to the Evaluator.

- The Algorithm provider also sends the labels needed to the Evaluator. These are labels corresponding to the predefined value V that will be checked in the circuit, the labels for the total randomness R, as well as labels corresponding to 1, in order to create NOT-gates out of XOR-gates.
- The Algorithm provider creates correlated randomnesses that add up to R, as well as shares of input labels, and sends each party their shares.

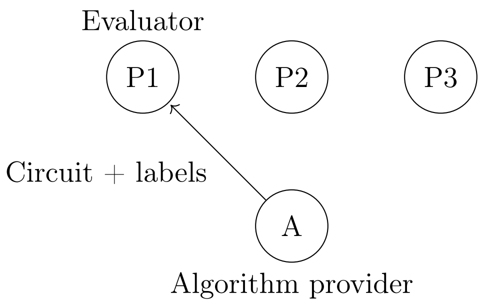

Figure 4.2: First the Algorithm provider sends the circuit and the labels for R, V and ones (1s) for the NOT-gates to the Evaluator.

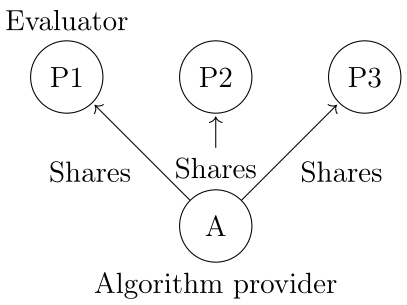

Figure 4.3: Then the Algorithm provider sends the correlated randomnesses and label shares created for each party.

The circuit

The circuit is comprised of two circuit parts, sub-circuits if one will. The first part is a subtraction circuit that will compute $Z - R$ and the second part is a comparision circuit that will decide if the result of $Z - R$ is equal to a value V. Both R and V are chosen before the circuit is sent to the Evaluator. The way that these values are practically chosen for the circuit is that the labels corresponding to the bits in these numbers are sent to the Evaluator and will be used together with the other input labels that the chain will produce. Therefore they are not per se "hard coded" into the circuit, instead they are taken as parameters, just as a suggestion in the paper [34].

The subtraction circuit is created just like a normal hardware subtraction circuit. Each bit of Z will be compared to the corresponding bit of R and the difference and carry out will be returned. A one-bit full subtractor is made out of seven gates. However, instead of using NOT-gates, as they are unavailable in the gabes library, XOR-gates are used that take as input the regular input and a one (1).

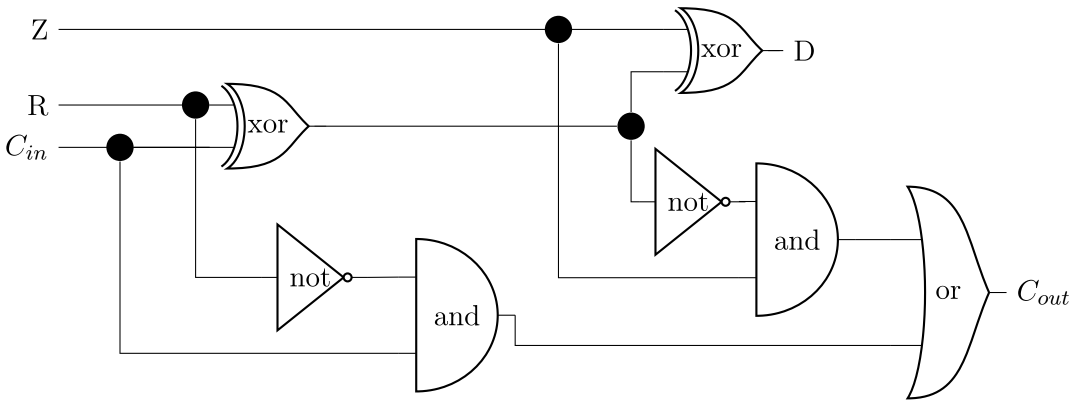

Figure 4.4: A one-bit full subtractor that computes $Z - R$. The circuit takes three inputs: one bit of Z, one bit of R and a carry in bit. It creates two outputs: the difference D and the carry out. Notice that some inputs and outputs are used multiple times in this circuit.

The figure above shows subtraction of only one bit. In order to compute subtraction with numbers of a greater bit length, one will have to cascade many one-bit subtractors. The carry out of a subtractor will be used as the carry in for the next. However, the carry in of the first subtractor is naturally zero, and the carry out of the last subtractor will tell if the number R could be subtracted from , i.e. the final carry out will be 1 if R was greater than Z.

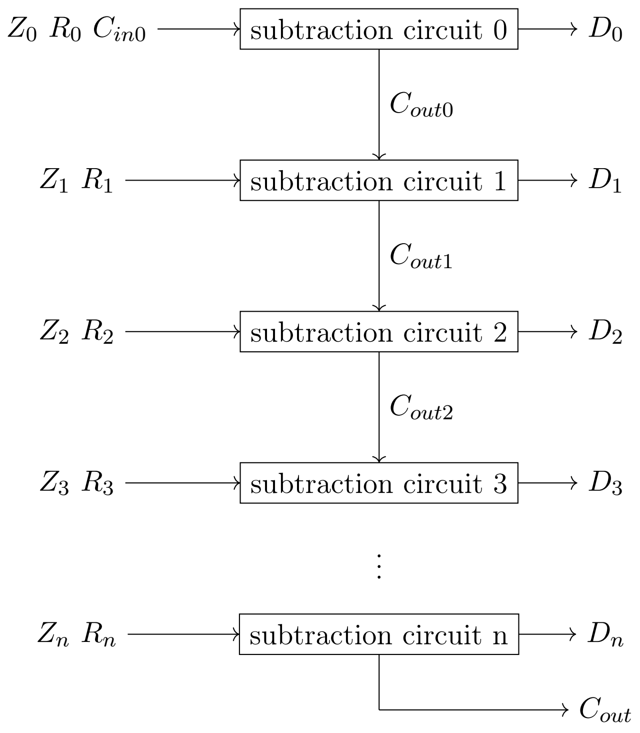

Figure 4.5: An n-bit subtractor can be created by cascading n one-bit subtractors. 0 is the position of the least significant bit.

An n-bit subtractor is the first circuit part of the whole circuit. It is realised by putting all the seven gates of each one-bit subtractor in an array and garble and ungarble them in the correct order. The inputs and outputs for each gate are located just as they are shown in the figures above, however remember that these are not physical attributes such as a metal wire, rather they are object attributes in the code.

The second circuit part is a comparison circuit. Each bit of the difference from the n-bit subtraction circuit will be compared to the corresponding bit in the value V that we want to compare against.

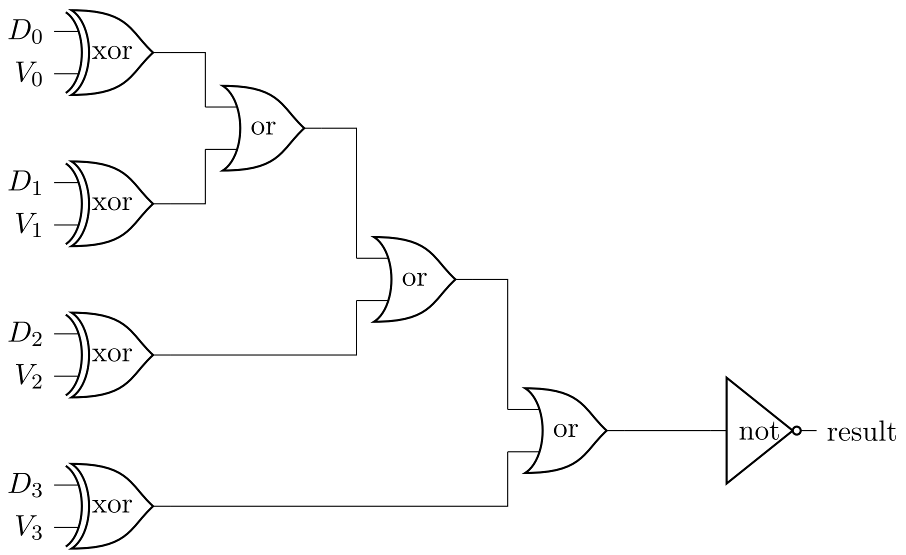

Figure 4.6: A 4-bit comparison circuit. Each pair of bits are compared using XOR-gates. OR-gates are then used to check if the first, or the second, or the third, or the fourth bit-pair were different. If one or more of the pairs didn't match, the last OR-gate will output 1. The NOT-gate in the end inverts the result, meaning that it will produce a 1 if the two numbers are the same, and a 0 if they are not.

This might be a good time to remind the reader that while in the figures above, and for all logical purposes, there are ones and zeroes passed through the system, this never happens in the code. Instead, labels are used as keys to decrypt only one possible combination that has already been calculated by the Algorithm provider. The garbled truth table of the encrypted output labels of each gate have been sent to the Evaluator, and while there are 4 possible options for each gate, only one is decryptable with the two input labels that the Evaluator gains for each gate. It is important to ungarble in the right order, as the Evaluator will only gain access to the label it will need for decryption of a gate by decrypting it at a previous gate. However, remember that the labels needed to start this process has also been sent to the Evaluator in the setup phase. The Evaluator will have access to the labels needed for R, V and the ones (1s) for the NOT-gates, but as it does not have access to the labels corresponding to the bits in Z it can not sucessfully ungarble the circuit until it receives them as well.

Correlated randomness and label shares

Both the randomness R and the labels corresponding to each bit in Z are cut up into pieces that are called shares. There are different ways of creating shares, but in this case additive shares are created. For example the number 100 could be made into three shares with values 50, 30, 20. Without all three shares no one would know what the total number is, in this way they are correlated. One way of doing this is to start with random numbers that are the shares and then add them together to create the total number, however this proved unpractical in code. The reason is that the numbers need to be of a fixed bit length, as the circuit is created before any calculations are made. For example, if one would like the total to be a random 32-bit number, just adding n random 32-bit numbers together would with an increasing probability create a total number of a greater bit length than 32. In order to avoid this situation, that proved to be quite prevalent while developing the code, a different tactic was used. A random 32-bit number N was created. Then n random and unique numbers within the interval 0-N are picked, and they are ordered in a list. The difference between the adjacent numbers in the list then becomes the shares. With some probability multiple shares could have the same value, however that is not a problem as they do not reveal anything about any other number.

4.1.2 Phase 1

In phase 1 of the protocol, each party sends to the next their value + their correlated randomness, along with what they received from the previous party. This is masking of the value v_i so that it is impossible to know what their actual value is. The algorithm provider does nothing in this phase.

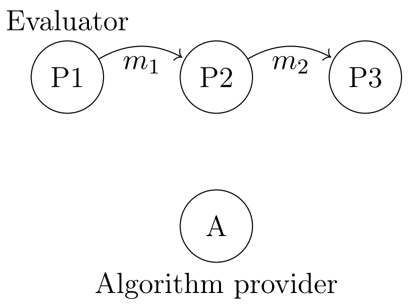

Figure 4.7: Phase 1. Note that $m_1 = v_1 + r_1$ and $m_2 = v_1 + v_2 + r_1 + r_2$

4.1.3 Phase 2

In phase 2 the last party in the chain computes the finalised Z that is the sum of each party’s value + the total randomness R. This is then sent in a backward pass through the chain where each time the label shares are added together. Each party has shares of both the True (1) and False (0) labels for each bit in the number Z. The label share that is chosen depends on the finalised bits in Z. If for example $Z = [0, 1, 0, 1]$, the label shares that are sent will be chosen accordingly by each party, as it is no secret if the share represents 0 or 1. Again, the Algorithm provider does nothing.

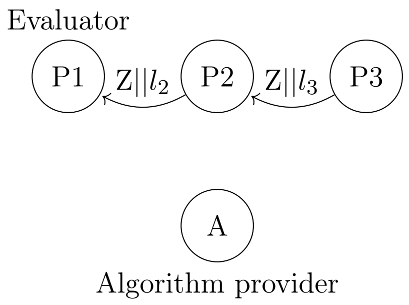

Figure 4.8: Phase 2. Note that Z is always the same, however with each message l increases as the shares are getting summed up.

In the end of Phase 2, the first party, the Evaluator, has both the number Z and all the complete labels corresponding to each bit in the number Z. The Evaluator now ungarbles the circuit which reveals the result.

4.1.4 Phase 3

In the last phase of the protocol, the result is passed through the chain. The Algorithm provider remains passive.

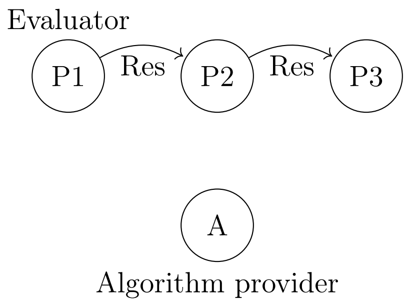

Figure 4.9: Phase 3. Distribution of the result.

4.2 Protocol 2

Protocol 2 was not implemented because of reasons discussed in chapter 7.

Chapter 5 Tests and Results

The three things that are tested are the BNC, latency and computational complexity of the protocols. All tests are performed for the number of parties ranging from 3 to 1000. The size of the numbers used in the protocol are 32 *bits* - the regular size of an unsigned integer in many programming languages. This regards the numbers Z, R, V. The labels and their shares are however of a size up to 32 *bytes*.

5.1 BNC tests and results

The purpose of testing the BNC is to verify that it is independent of the numbers of parties in the protocols. As explained in [34], the BNC regarded is for the messages sent through phases 1 to 3. A reminder is that the BNC is defined in Section 2.3.4 as the maximum number of bits any party in the protocol sends or receives.

The BNC of the online phase of the protocol, namely phases 1-3, is tested through intercepting each message sent to and from each party in the chain. The sum of the number of bits of all messages sent to each party is recorded, as well as the sum of bits of all messages sent from each party. From this information a number of metrics can be derived such as the median, minimum and maximum number of bits sent to and from any party in the protocol. The maximum is the BNC, however it takes little additional effort to derive the other information.

In each graph below, the red line shows bits sent and the blue line shows bits received. The BNC is defined as the maximum number of bits sent or received by any party in the protocol. The maximum number of bits is all that is required to know the BNC of the protocol, however, while at it, there is very little effort into deriving some other metrics from the tests and they can give some interesting insights.

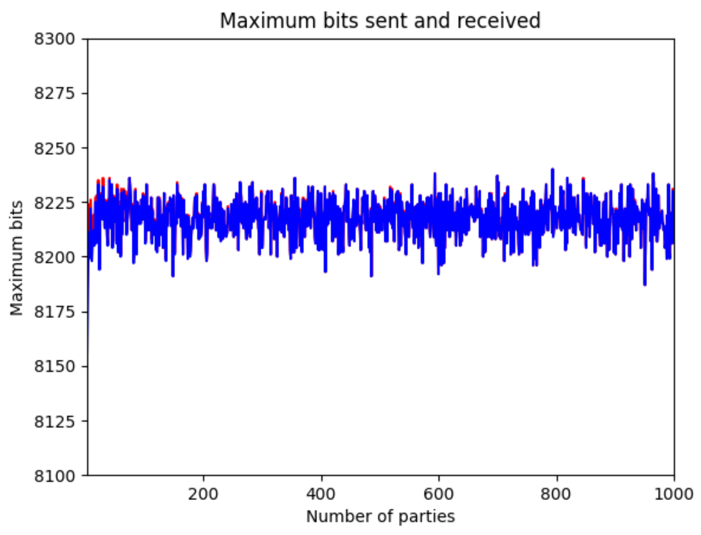

Figure 5.1: The BNC of the first protocol can be seen above. The maximum number of bits sent, seen in red, and the maximum number of bits received, in blue, are both at fairly equal levels, as well as keeping around 8220 bits, unaffected by the number of parties in the protocol.

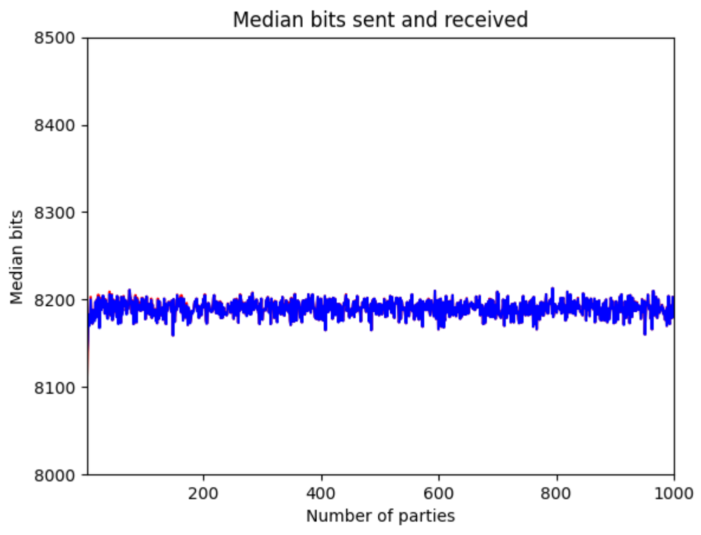

Figure 5.2: The median number of bits sent and received in the protocol can be seen above in red respective blue. As expected, the median is slightly lower than the maximum, and still unaffected by the number of parties in the protocol.

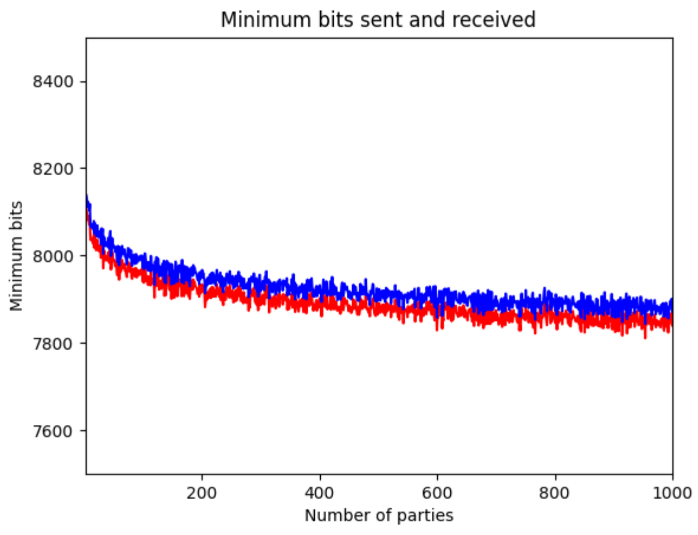

Figure 5.3: The minimum number of bits sent, in red, and received, in blue, except the outlier numbers for the first and last party who naturally sends and receives less than the other parties. This seems to be the only case where the number of bits sent and received are affected by the number of parties in the protocol. This is explained by the fact that the probability of a label share to become smaller increases as the number of parties increases in the protocol. If a label share is a smaller number, it naturally requires fewer bits to be sent.

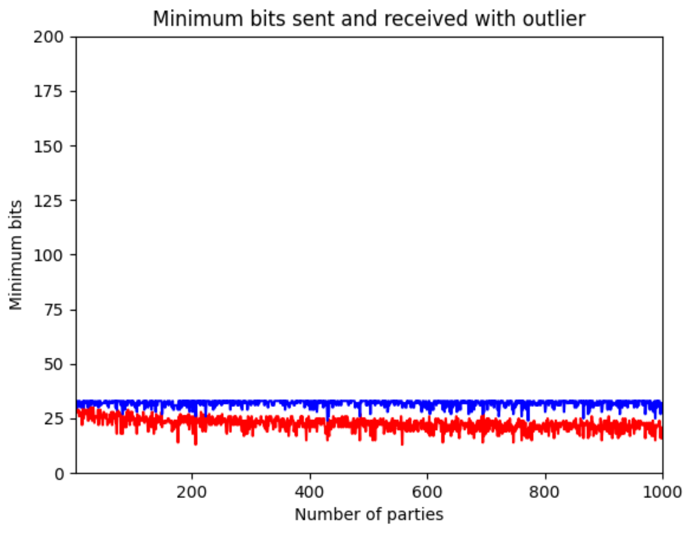

Figure 5.4: The number of bits sent by the first party, and the number of bits received by the last party, will be drastically less than for any other party in the protocol, which is why they are outliers. The reason is that the first party never sends their 32 different label shares needed to compute the whole, it already has them. The last party never receives the 32 label shares from any party as it doesn’t need them.

5.2 Latency tests and results

The purpose of latency testing is to see in what way the latency is behaving as more parties are added to the protocol. Naturally one would assume that with more parties, the total latency would increase. However, finding what kind of relation links the number of parties to the latency is of importance to understand the overall behaviour of the protocol in real life.

Latency is tested through measuring the overall time it takes for the protocol to run in the online phase. The results can be seen in the graph below.

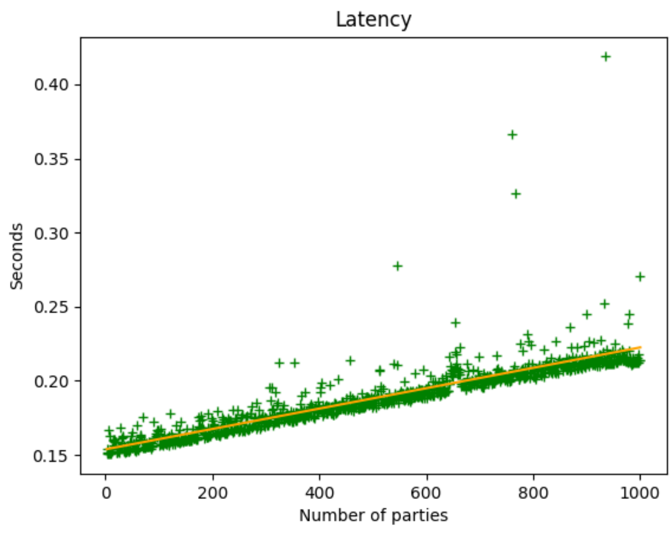

Figure 5.5: Latency of the protocol depending on the number of parties. The orange line represents linear regression based on the latency values, producing a straight line that fits the data points fairly well.

While there are some outlier values of the latency measurements, most values seem to fit in quite well into a linear equation. A standard linear regression function was used to create the line that shows the best fitting line for a function of the first degree. The slope of the line tells us how much the addition of one party to the protocol adds to the latency. Each run of the simulation will have varying results as the run time can be affected by what other things the computer has to do at that time, and therefore 10 runs were performed to get a sense of what variation there could be.

nr simulation	Slope of line
1	9.103425288495664e-05
2	8.372617713488851e-05
3	6.488357822920954e-05
4	6.504099920718382e-05
5	7.235784405534714e-05
6	7.437294606410753e-05
7	6.541773334639031e-05
8	6.668103578013845e-05
9	6.430970086696766e-05
10	6.54645806888198e-05

Table 5.1: Slope of the line for 10 simulations. This is what each party adds to the run time on average for each simulation.

5.3 Computational complexity measuring and results

The purpose of measuring computational complexity is to better understand how the workload increases depending on variables such as the number of parties in the protocol as well as the size of the numbers used. In this protocol, the first party will bear the biggest burden as it is ungarbling the circuit.

Computational complexity is calculated through computing what operations each party has to perform through the protocol, and seeing in what fashion the number of computations would increase and what it depends on.

There are three different cases in the protocol regarding the computational complexity. The first case is regarding the first party, which naturally has a higher workload than the other parties, as it does the ungarbling of the circuit. The second case is the last party, who will do the least computation. The last case regards all the other parties in the protocol, the middle parties.

5.3.1 The first party

The first party bears the heaviest burden. Below on the left, the process is explained in words, and on the right an expression for the computational complexity is built.

Starting off, it computes one addition, in order to be able to send $v_1 + r_1$ to the second party.	$1\ Add$
	$+$
As it receives the result of the second chain pass, it computes one addition per label, meaning one addition per bit in the number Z. In this case it is 32 additions. let b denote the number of bits.	$b\ Add$
	$+$
The ungarbling process depends on the encryption scheme and garbling scheme. Using standard ungarbling, the party will try to decrypt each entry in the garbled truth table until it succeeds. In the worst case it will have to try four decryptions per gate. Let g denote the number of gates.	$(4g)Dec$
	where
The number of gates is dependent on the number of bits in the numbers Z and V. The subtraction circuit is comprised of seven gates per bit. The comparison circuit is made of $2b$ gates. In total the circuit includes $9b$ gates.	$g = 9b$

The additions that are done in the beginning are of minor importance compared to the decryption process. In the worst case, decryption is performed 36 times per bit in the number Z. The result is that the computational complexity does not depend on n - the number of parties in the protocol - instead it could be described as $O(36b)$ in terms of numbers of decryptions made that depends on the number of bits in Z. This is a linear case making it possible to exclude the constant term and describe it as just

$$O(b)$$

The number Z and the number of bits b is not dependent on the number of parties n either, though logically it is reasonable to assume that $Z > n$ for it to be possible for each party to cast a vote with the value 0 or 1. Luckily, b does not have to be larger than n, but it should hold that $2^b > (n + R)$. In our case it is true that $2^{32} > (1000 + R)$, where we use 32 bits, up to 1000 parties in the tests and a total randomness R.

5.3.2 The last party

The last party only handles 2 additions after it receives the result from the first pass in the chain, in order to compute Z. This computational complexity can be described as $O(1)$ as it will always be the same.

5.3.3 The middle parties

The middle parties performs 2 additions during the first pass of the chain in order to compute $z + v + r$. In the second pass it performs b additions, one for each label share. The result is $(2 + b)$ additions. This computational complexity is also linearly dependent on the number of bits b and is notated $O(b)$.

Chapter 6

Discussion and Conclusions

The two protocols are discussed separately below. First, the results regarding the first protocol are discussed. Then, the second protocol and the shortcomings that led to it not being implemented is discussed.

6.1 Protocol 1

The first protocol was implemented and test results could be obtained. For each of the three results for BNC-testing, latency-testing and computational complexity testing, a standalone discussion is presented, after which a complete conclusion is drawn.

6.1.1 BNC

The experimental results show that the BNC of protocol 1 is independent of the number of parties in the protocol and remains at a fairly stable level through the simulations. The slight variation seen in Figure 5.1 is explained by the fact that not all label shares are always the exact same amount of bits, meaning that two different runs can produce different results as the labels and label shares are sampled at random. The median number of bits sent and received is as well independent of the number of parties in the protocol, and remains slightly lower than the BNC - the maximum number of bits sent or recieved. In addition, the minimum bits sent and received, excluding the outlier cases, seems to drop as more parties are added to the protocol. This is explained by the fact that with more parties and label shares the probability of a share to become smaller increases. This is good if one is only regarding the cost of communication. For security purposes this feature is however debatable. Figure 5.4 shows the two cases where the minimum bits sent and received includes the outlier cases. The last party in the protocol will receive less bits than any other party as it does not have to receive any label shares. The first party will similarly not have to send any label shares. These shares evidently make up the most of the communication happening. Using label shares like this enables the the avoidance of Oblivious Transfer, which would have served the same purpose but added to the BNC.

6.1.2 Latency

The latency of the protocol is linearly increasing with each party added to the protocol, which is expected. A linear regression was used to find a slope that would represent how much each new party added to the total latency on average. This turned out to be around 64-91 micro seconds, deriving this information from 10 simulations. The total latency stayed under half a second for all simulations. These results show that the protocol does not inherently increase the latency out of proportion as more parties are added. However, running this protocol on multiple machines across the world could have produced different results as in that case the latency would also depend on the physical distance between the machines and the network speed with which it is operating. With bad luck the adjacent parties in the chain could be located physically very far from each other.

6.1.3 Computational complexity

Both the first party and the middle parties has a computational complexity of $O(b)$, while it is apparent that the first party does bear a bigger burden. The interesting part is that the complexity is linearly increasing, and dependent on the number of bits handled, unaware of the number of parties in the protocol. The linearity is good news and the dependancy on the number of bits is expected. The last party has a computational complexity of $O(1)$ however, which opens up for possible usages of this particular feature. The computational complexity of the protocol is better for the last party than for the others, however all has a linear or constant relation to the number of bits b in the number Z and no relation to the number of parties n in the protocol.

6.1.4 Threats to validity

The randomness in the tests regarding the BNC and latency are possible threats to the validity of the tests. The BNC is fluctuating slightly, and while this could be mitigated by for example zero padding the messages sent, it is deemed unnecessary as it is something that is taken care of by most communication protocols that are used in online settings. Additionally, there is no benefit in zero padding from a security perspective, while it would increase the communication. In the simulations for latency some randomness is seen as the computer running the simulations will have other tasks to perform at the same time. In order to minimise this problem the simulations were run with as little interference from other programs and tasks as possible, as well as creating the linear regression that would show how the simulation performed typically, taking into account the small differences that there can be between runs. For the computational complexity there was no tests performed, only mathematical estimations.

6.1.5 Final conclusions

All three aspects that have been examined have produce expected results where the BNC, latency and computational complexity has either stayed on stable levels or increased linearly. The chain-like communication pattern was successful in its main purpose to make the BNC independent of the number of parties in the protocol.

6.2 Protocol 2

The second protocol was deemed to complicated and inefficient for a reasonable implementation. The problem regarded the fact that garbled circuits were to be used for both encryption and decryption, meaning that boolean circuits performing these mathematical operations had to be constructed. This might not seem to be a big issue at first, but there are several reasons why this is not very simple.

First, let’s look at the purpose of garbled circuits and some expectations on the technique. A garbled circuit can not reveal any partial results. If it did, it would not be fit for MPC as only the ouput of the computation is to be learned. An example where this is problematic is during the encryption process in the second protocol. There are several additively homomorphic encryption schemes that could be implemented, see Table 3.2, but arguably the easiest ones would be El Gamal or Paillier. The GC is supposed to encrypt a 1 if the index of the party is the same as the one that was chosen, and 0 otherwise. The index is represented by labels used by the GC, thereby preserving privacy of the actual value, and can be easily compared to the party’s index through the same circuit part as was used in protocol 1. However, when the chosen index and the party index have been compared, the problems begin. Let’s look at the encryption process of Paillier

$$c = g^m \cdot r^n \bmod n^2$$

In this case, m would be either 0 or 1. Exponentiation rules states that

$$g^0 = 1$$

$$g^1 = g$$

This is technically an if-statement, the computation will look different depending on what the input is. There will be a label used instead of a number, but how can you know if it is representing a 1 or a 0? Hopefully, you can’t. If you could, this would reveal the information the protocol is trying to preserve, namely what index is chosen. Therefore, this label will have to go down two different paths from now on. There will be one computation for g^0 and one for g^1, and both will have to be performed. Maybe there could be a way to select only one of them, but then that would reveal which path the computation is taking and thereby reveal this partial result which is very clearly in the need to be secret. The computation has been split into two paths, and because no gate can be used twice, even if the computations are the same from here on, they have to be replicated in each path. This is only the beginning of the problem.

Imagine now that a modular operation has to be performed. What does this look like in a boolean circuit? Most likely, the easiest solution is to check whether the value is above the threshold n^2, for Paillier, and in that case subtract n^2 from the value. Then a new check has to be performed in order to see whether the result is still over the threshold or not, and repeat the process as necessary. With each check, a new path will have to be created and from there on the process has to be replicated for each path.

Let's look at exponentiation. The most straight forward way to construct a boolean circuit for his operation is to set a number of multiplication circuits in series. Then of course you have to know how many you need, as they can't be used twice, and if the exponent is secret, this is a bad idea. During the decryption process of both Paillier and El Gamal the secret key for decryption is used as an exponent.

Decryption with Paillier, where λ and μ makes the secret key:

$$m = L(c^{\lambda} \bmod n^2) \cdot \mu \bmod n$$

More efficient ways of constructing a exponentiation circuits are possible, but this still entails that one has to know beforehand how many multiplications has to be performed or create new paths for each if-statement in the algorithm. A way of battling this is to create a circuit that computes all possible combinations. This would be extremely inefficient and costly, both computationally and memory-wise, as one can imagine. If one were to do this, however, when all paths are taken and everything is done, how will one know which one is the result? Also, the results of all these computations will reveal too much information, one could probably derive the secret key for example from the sheer amount of information that has been computed. A way of tackling this problem would be to multiply the result of each path with either a 0 or 1 depending on if it is the correct result or not, and take the sum of all these products. The end result would then only be the correct result and there would be no other partial results that can provide any excess information. This is, funnily enough, the functionality of the second protocol itself.

Clearly, creating garbled circuits with these functionalities are, while not technically impossible, quite difficult. A straight forward approach is not efficient enough for an implementation to be created within the time frame of this thesis work.

Chapter 7 Concluding remarks

There are a few concluding remarks for both of the protocols. For the first protocol they can mainly pose as inspiration for future work, while for the second protocol it is a note on a design choice.

7.1 Remarks regarding protocol 1

The security of the protocol is something that could be examined and considered for improvement in future work. As theoretically proved, the protocol is secure against passive adversaries, however there is no way of verifying that any party sends information that is acceptable. The current implementation accepts for example inputs that are many bits long, while perhaps it was only sought after that each party submits only one bit as their value. In the use case presented in the introduction, each party can only vote yes (1) or no (0), but how is this supposed to be verified? How the protocol should handle numbers that exceed a certain limit, perhaps through modulo operations, is not considered. In any case, verifying that what has been received is acceptable is not possible in the current proposal. An adversary could, while following the protocol specification, mess things up. In future work regarding this protocol, this is an aspect to work on.

Another aspect to examine is the setup phase of the protocol. In the paper [34] there is no clear place for the entity that garbles and sends the circuit, making it external to the protocol as of now. This raises the question whether the garbler should be a party within the chain of the protocol, or not. The setup phase is neither taken into account during the tests as it is not addressed as such within the paper. A reason to not have the garbler be an actual party in the protocol is that it gives too much power to that party. As all labels ever needed and used in the circuit are created through the process of garbling the circuit, the party that does the garbling can just save any information about them. Even though the complete labels are not shared with anyone except the first party, a garbling party could have saved them and perform calculations with them based on the knowledge of Z. Partial information such as what the sum of all parties input is can be derived. Relating to the IEEE recommended practice for SMPC [1] there should be a bit more thought about the different roles of the parties in the protocol before going towards practical use.

On the note of different roles for parties in the protocol, there is one crucial role missing. Who is the *Task initiator* and *Coordinator*? In the implementation this question is answered by having a simple object keeping track of other objects that are the parties in the protocol, and making sure that they communicate the correct messages in the right order. However, in real life there would have to be an additional step in the protocol in which parties decide to join the protocol and get some kind of position in the chain. This process could also have an effect on the latency of the protocol.

7.2 Remarks regarding protocol 2

Described in [34] there are two ways to provide the parties with the encryption of 0 or 1 that they need. Either a table with an encryption per party could be given to and stored by each party, or the garbled circuit approach can be used. The garbled circuit is opted for by the authors, however this is probably not the best approach in many cases. The reason is that the garbled circuit that has to be constructed in order to perform the encryption will contain a very large number of gates, and each gate will technically just be a table of 4 ciphertexts. In order for this to be more efficient than a lookup table of encryptions, there will have to be a larger number of parties participating in the protocol than 4 times the number of gates, plus some for all the labels and additional information needed in order to know in what order the gates should be ungarbled.

References

[1] Ieee recommended practice for secure multi-party computation. *IEEE Std 2842-2021*, pages 1–30, 2021. doi: 10.1109/IEEESTD.2021.9604029.

[2] Garbled circuit, 2022. URL `https://en.wikipedia.org/wiki/Garbled_circuit`.

[3] AndrewQuijano. Homomorphicencryption · github. `https://github.com/AndrewQuijano/Homomorphic_Encryption`.

[4] D. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. Pagter, N. Smart, and R. Wright. From keys to databases—real-world applications of secure multi-party computation. *Computer Journal*, 61:1749–1771, 12 2018. doi: 10.1093/comjnl/bxy090.

[5] J. Auterson. go-garbled. `https://github.com/JoelOtter/go-garbled`, 2015.

[6] D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-dnf formulas on ciphertexts. In J. Kilian, editor, *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30576-7.

[7] E. Boyle, A. Jain, M. Prabhakaran, and C.-H. Yu. The Bottleneck Complexity of Secure Multiparty Computation. In I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-076-7. doi: 10.4230/LIPIcs.ICALP.2018.24. URL `http://drops.dagstuhl.de/opus/volltexte/2018/9028`.

[8] M. Brenner. yao. `https://github.com/hcrypt-project/yao`, 2013.

[9] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, Jan 2000. ISSN 1432-1378. doi: 10.1007/s001459910006. URL `https://doi.org/10.1007/s001459910006`.

[10] Z. Cao, C. Huang, and Y. Li. A study on the improvement of computation, communication and security in garbled circuits. In *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)*, pages 609–617, 2021. doi: 10.1109/ICSP51882.2021.9408745.

[11] J. B. Clarkson. Dense probabilistic encryption. In *In Proceedings of the Workshop on Selected Areas of Cryptography*, pages 120–128, 1994.

[12] S. Collins. *The hunger games trilogy.* Scholastic, New York, NY, 2014. ISBN 9780545670319.

[13] Cryptovote. Damgard-jurik. `https://github.com/cryptovoting/damgard-jurik`, 2019.

[14] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In K. Kim, editor, *Public Key Cryptography*, pages 119–136, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44586-9.

[15] C. Data61. Python paillier library. `https://github.com/data61/python-paillier`, 2013.

[16] ELECTRIC COIN COMPANY. Zcash. URL `https://z.cash/`.

[17] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. doi: 10.1109/TIT.1985.1057074.

[18] emp toolkit. emp-tool. `https://github.com/emp-toolkit/emp-tool`, 2022.

[19] ENCRYPTO. Motion. `https://github.com/encryptogroup/MOTION`, 2020.

[20] S. Goldwasser and S. Micali. Probabilistic encryption amp; how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 365–377, New York, NY, USA, 1982. Association for Computing Machinery. ISBN 0897910702. doi: 10.1145/800070.802212. URL `https://doi.org/10.1145/800070.802212`.

[21] S. Halevi, Y. Ishai, A. Jain, E. Kushilevitz, and T. Rabin. Secure multiparty computation with general interaction patterns. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ITCS '16, page 157–168, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340571. doi: 10.1145/2840728.2840760. URL `https://doi.org/10.1145/2840728.2840760`.

[22] jianyu m. damgardjurik. `https://github.com/jianyu-m/damgard_jurik`, 2017.

[23] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition.* Chapman Hall/CRC, 2nd edition, 2014. ISBN 1466570261.

[24] K. Koç, F. Özdemir, and Z. Ödemiş Özger. *Partially Homomorphic Encryption.* Springer International Publishing, 2021. ISBN 9783030876289. URL `https://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=cat02271a&AN=atoz.ebs29868464e&site=eds-live&scope=site`.

[25] T. M. Lab. encryptionschemes.paillier. `https://github.com/TNO-MPC/encryption_schemes.paillier`, 2022.

[26] N. C. R. Labs. Paillier threshold encryption scheme implementation. `https://github.com/niclabs/tcpaillier`, 2020.

[27] LatencyUtils. glatencyutils. `https://github.com/LatencyUtils/LatencyUtils`, 2021.

[28] multiparty. Accessible and scalablesecure multi-party computation, . URL `https://multiparty.org/`.

[29] multiparty. multiparty · github. `https://github.com/multiparty`, .

[30] multiparty. jigg. `https://github.com/multiparty/jigg`, 2021.

[31] D. Naccache and J. Stern. A new public key cryptosystem based on higher residues. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, CCS '98, page 59–66, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130074. doi: 10.1145/288090.288106. URL `https://doi.org/10.1145/288090.288106`.

[32] N. Navarro. gabes. `https://github.com/nachonavarro/gabes`, 2018.

[33] T. Okamoto and S. Uchiyama. A new public-key cryptosystem as secure as factoring. In K. Nyberg, editor, *Advances in Cryptology — EUROCRYPT'98*, pages 308–318, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-69795-4.

[34] C. Orlandi, D. Ravi, and P. Scholl. On the bottleneck complexity of mpc with correlated randomness. In G. Hanaoka, J. Shikata, and Y. Watanabe, editors, *Public-Key Cryptography – PKC 2022*, pages 194–220, Cham, 2022. Springer International Publishing. ISBN 978-3-030-97121-2.

[35] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48910-8.

[36] A. Poon. Bgn. `https://github.com/anna138/BGN`, 2019.

[37] P. Rindal. Ivory-runtime. `https://github.com/ladnir/Ivory-Runtime`, 2019.

[38] S. Servan-Schreiber. paillier. `https://github.com/sachaservan/paillier`, 2020.

[39] E. Songhori. Tinygarble. `https://github.com/esonghori/TinyGarble`, 2019.

[40] X. Wang. Flexsc. `https://github.com/wangxiao1254/FlexSC`, 2018.

[41] A. C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982. doi: 10.1109/SFCS.1982.38.

[42] A. C.-C. Yao. How to generate and exchange secrets. *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986.

[43] ziyao002. Threshold-paillier-with-zkp. `https://github.com/ziyao002/Threshold-Paillier-without-Trust-Dealer`, 2019.