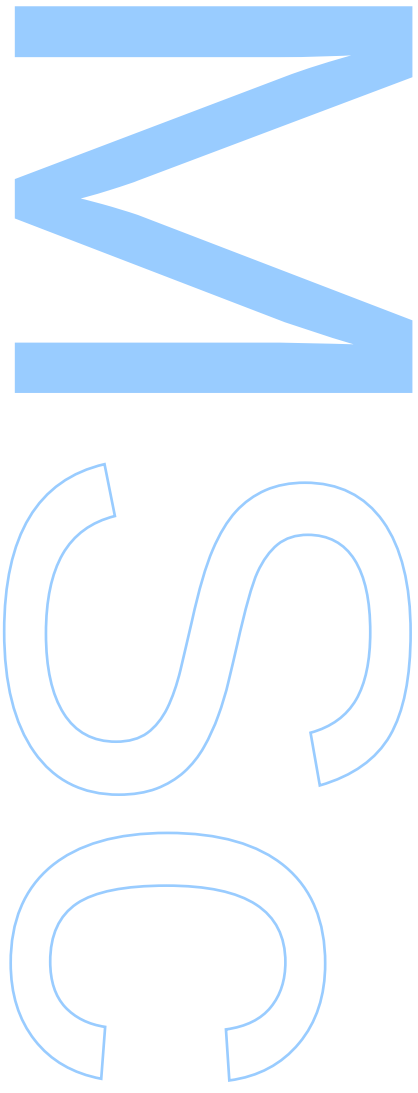
Efficent Verified MPC

Manuel Luís Magalhães Duarte Correia
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2021

Supervisor
Manuel Bernardo Martins Barbosa, FCUP



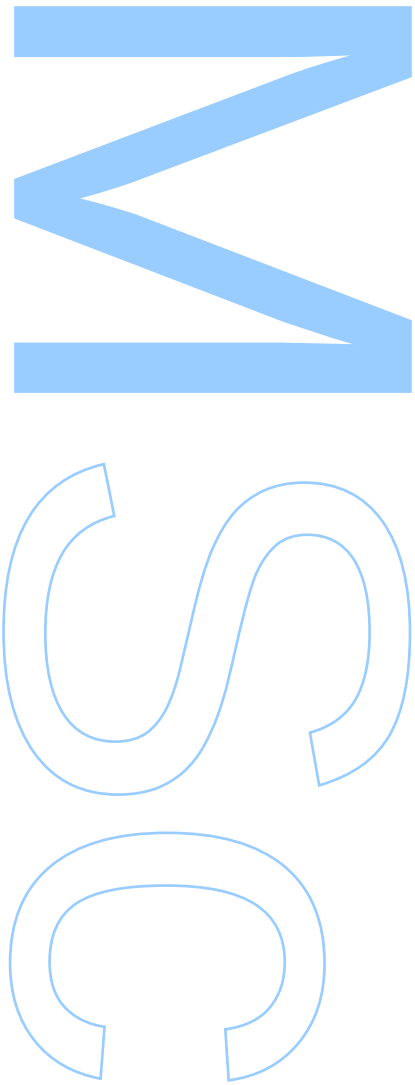
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Acknowledgements

I would like to thank my supervisor Manuel Barbosa who made this work possible and for all the availability and help throughout the year. Along with my supervisor, I would also like to thank José Carlos Bacelar Almeida, Hugo Pacheco, Tiago Oliveira, Vítor Manuel Parreira Pereira, Jorge Sousa Pinto and Simão Melo de Sousa for the help in understanding the protocols and concepts I wasn't familiar with, for being friendly and welcoming and for all the opportunities given, including the scholarship. I also would like to thank my father, to whom I turned to for reassurance about my work many times, and my mother who has always supported me. Finally, I'd like to thank my brothers and all of my friends for being a source of constant support throughout my life.

Abstract

Secure multi-party computation (MPC) is an approach to cryptography associated with a group of parties' collaborative computation of a function, with the primary purpose of keeping the inputs of this function a secret. In this manner, each party's data can be protected while a common output is reached.

These protocols allow for the development of secure technology and decentralized systems such as online auctions, electronic voting and other applications requiring the protection of the input data of the participants involved.[15]

The main objective of this work is to create an efficient and passively secure implementation of a MPC protocol that enables the computation and evaluation of secret shared data. Furthermore, the implementation of this protocol is to able to be used as a component in generic zero-knowledge proofs following the MPC-in-the-head paradigm.

This work also involves the first Jasmin language MPC implementations, a language developed for the creation of high-speed cryptographic programs, which allows for the optimization and easy formal verification of the protocol's code as secure.[1]

The main implementation developed is based on an approach to passively secure MPC described in Ueli Maurer's paper *"Secure multi party computation made simple paper"*[18] which was adapted to 3 or 5 parties with the addition and multiplication as the major available operations, while allowing for publicly known values. Besides this, an implementation for Boolean circuits based on the Goldreich-Micali-Wigderson Protocol was also created.

The current work describes all of the steps involved in the development of all the MPC implementations created, detailing how they work, how to use and integrate them into new systems, and testing, as well as discussing the results obtained.

Resumo

Secure multi-party computation (MPC) é uma abordagem de criptografia associada à computação colaborativa de uma função por um grupo de participantes, com o objetivo principal de manter os inputs dessa função secretos. Desta forma, os dados de cada parte podem ser protegidos enquanto um output comum é alcançado.

Esses protocolos permitem o desenvolvimento de tecnologias seguras e sistemas descentralizados como leilões online, votação eletrônica e outras aplicações que requerem a proteção dos dados de entrada dos participantes envolvidos.[15]

O objetivo principal deste trabalho é criar uma implementação eficiente e passivamente segura de um protocolo MPC que permite o cálculo e avaliação de dados secretos compartilhados. Além disso, a implementação deste protocolo terá de ser capaz de ser usada como um componente em provas genéricas de conhecimento zero seguindo o paradigma MPC-in-the-head.

Este trabalho envolve também as primeiras implementações MPC da linguagem Jasmin, uma linguagem desenvolvida para a criação de programas criptográficos de alta velocidade, que permite a otimização e fácil verificação formal do código do protocolo como seguro. [1]

A principal implementação desenvolvida é baseada numa abordagem para proteger passivamente o MPC descrito no artigo de Ueli Maurer *"Secure multi party computation made simple paper"*[18] que foi adaptada para 3 ou 5 participantes, com a adição e multiplicação como as principais operações disponíveis e que também permite valores publicamente conhecidos. Além disso, foi criada uma implementação para circuitos booleanos baseada no protocolo Goldreich-Micali-Wigderson.

O presente trabalho descreve todas as etapas envolvidas no desenvolvimento de todas as implementações de MPC criadas, detalhando como funcionam, como utilizá-las e integrá-las em novos sistemas, como as testar, e discute os resultados obtidos.

Contents

List of Tables

List of Figures

Code Block List

Acronyms

MPC Multi-Party Computation

OT Oblivious Transfer

GMW Goldreich-Micali-Wigderson Protocol

Smult Simple Multiplication

Chapter 1

Introduction

As the use of decentralized and cloud-based systems grows, as does the need for privacy, new problems and situations requiring new practical security solutions and approaches emerge. Situations involving multiple acting participants that wish to protect and guarantee the security of personal critical information are a common example of the problems that these security and privacy concerns give rise to.

Secure multi-party computation (MPC) can be used as a solution to these types of scenarios. A secure MPC protocol enables groups of multiple parties to jointly compute or execute operations while keeping every participant's inputs private and secure. There is no third party involved in these protocols besides the participants, and all of the participants engage in the computations.

This work entails the investigation, design, and development of secure MPC implementations that are efficient, can be integrated into multiple systems, and are verifiably secure. All development, testing, and analysis details will be documented so that the strategies and ideas used can be re-applied and reused in different MPC contexts and so the implementations developed can be more easily integrated into different systems.

1.1 MPC Applications

As previously stated, MPC can be used in situations where mutually distrusting parties are attempting to cooperatively compute a function using data that must be kept secret. The scenarios that follow will provide examples of how MPC can be used as a practical solution.

Yao's Millionaires Problem is a classic example for generalizing multi-party computation problems. It is explained as follows: "Two millionaires want to know who is the wealthier of the two. They do not, however, wish to learn how wealthy the other is or any other information about their peer." While this application is not very practical, it does serve as a simple example of the types of problems that secure MPC can solve.

Online auctions are one of the first discussed applications of secure MPC[16], as they raise

numerous security and privacy concerns and involve multiple parties. MPC in this situation, in addition to allowing bidders to protect their private information from auction organizers, can guarantee bid non-repudiation. Many types of online auctions require the highest bidder's information to be hidden while still displaying the highest bid, which is easily accomplished using secure MPC.

Electronic voting is another widely discussed application of MPC[16]. This is a controversial topic in computer security, as it is something very hard to practically execute without major vulnerabilities or ways to sway the vote. Nonetheless, secure MPC lends itself well to this, because counting votes is essentially a jointly computed addition function. The main issue with simply using an MPC protocol by itself is proving that a specific person participated in the voting (something that can be resolved using Zero Knowledge Proofs). Secure MPC, in either case, can be a very important and significant component in this area.

As privacy concerns about data collection and processing grow, it becomes more difficult to collect and use information for machine learning or other purposes.The main concern with mass data collection, and rightly so, is how this data can be used against individuals and how there are few guarantees about what happens to the information collected. Secure MPC can provide data owners with assurances that those who wish to process it will not use it for any other purpose than the one agreed upon at the outset[16]. Surveys that require potentially sensitive data such as salaries or medical history, for example, can be conducted with greater security guarantees for the individuals who own this data. On a larger scale, large real-world data sets collected during the use of a system can be used by a third party to train models without the data owners revealing any information.

Another smart MPC application that is being utilized is as a component in key management services[16]. This is accomplished by splitting a key into several pieces, which are then distributed to various parties. When the key is required, secure MPC is executed with all parties involved, computing a joint function that includes the cryptography operation involving the key. This safeguards the key by requiring an attacker to gain control of all key segment owners. While dividing a key into multiple pieces is not a novel idea (many hardware security modules and security systems use this tactic), the key is frequently rebuilt to perform the cryptographic operation, allowing for a single point of attack. While secure MPC may not be suitable for use in every system due to efficiency or other factors, it can be used to protect a key throughout its entire life cycle.

A secure MPC application that is currently being researched is used as a component in Zero Knowledge proofs. A zero knowledge proof, by definition, allows someone to convince a third party of an assertion without revealing any additional information other than the fact that the assertion is true. Secure MPC can be used to protect and keep information secret while creating a proof. The MPC-in-the-head paradigm[22] is an example of MPC being used in this manner.

1.2 Objectives and Scope

As was described, secure MPC is a broad topic that can be applied to many circumstances, and while the implementations developed aim to be adaptable to a variety of situations, it is still necessary to specify and prioritize the focus of this work. As such, the primary goal of this work is to create a secure MPC implementation that can be used as a component in generic zero-knowledge proofs based on the MPC-in-the-head paradigm.

To accomplish this, while the protocols and strategies developed will be presented in such a way that they can be used for any number of parties, the primary implementations will use three and five parties.The publication Machine-checked ZKP for NP-relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head[4] delves deeper into this zero knowledge paradigm using implementations developed during this work.

Furthermore, the presented implementations will be the first developments of MPC in the Jasmin language, a cryptography and security focused low level language that will be described in greater detail in the following chapter.

1.3 Structure of this document

The current work will first describe how secure MPC and related technologies work, namely the two protocols implemented described in Ueli Maurer's Secure multi-party computation made simple[18] and the Goldreich, Micali, and Wigderson (GMW) protocol[16]. Other techonologies such as secret sharing[17], Oblivious Transfer[20] and the jasmin language[1] will also be detailed.

This is followed by examples of more practical implementations of secure MPC and related technologies, such as MPC deployments and working implementations of the applications described above.

Following that, descriptions of the two main implementations created will be presented including challenges encountered, design choices, and how they can be used. The first is Maurer's MPC, which is followed by the GMW implementation. Topics covered include how the code was organized, how messages are created and processed, how memory is handled, and how the main code should be used.

Finally, the outcomes of these implementations, as well as how they were tested, are discussed. These topics include how the implementations were analyzed and tested, formal security verification, efficiency benchmarks, and examination.

1.4 Publications

The following publications were published as a result of this work, where the second is the full version of the former: Machine-checked ZKP for NP relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head[5][4] submitted and accepted to the ACM Conference on Computer and Communications Security 2021[1].

[1]The paper can be found as accepted in https://www.sigsac.org/ccs/CCS2021/accepted-papers.html

Chapter 2

Background

A significant challenge in implementing a secure and efficient MPC protocol is learning about how MPC works, its components, the various protocols that exist, and the differences between them. This chapter will describe the background and important information required to fully comprehend the implementation and the work completed.

2.1 Secret Sharing

A secret sharing method is a key component of secure multi-party computation. These methods entail splitting a value that must be kept secret into several shares (sharing), and then reconstructing the secret using some or all of these shares (unsharing). These shares are created by a dealer with the secret and distributed among n participants in such a way that only defined subsets of these parties can retrieve the secret. In an MPC protocol if all values are secretly shared no participant has control over any single secret value.[7]

The way a value split into several shares is reconstructed and distributed may vary case-to-case. In certain applications it might be useful for all the shares to be required to retrieve the original secret. In others, there might be a need for each participant to have more than one share associated with them. This distribution is called a secret sharing scheme.

While each MPC protocol may have various criteria for its secret sharing methods, the sharing method is required to be homomorphic in most cases. The shares in these methods are generated using homomorphic encryption, which allows them to be used in computations in such a way that, when they are unshared, the resulting value is identical to the output of the same computations on the original secret.

If we imagine the MPC protocol operating over a circuit of computations, each computation, also known as a wire or gate, will have a set of shares distributed among the participating parties that, when unshared, will reveal the wire's output. Because of the sharing method's

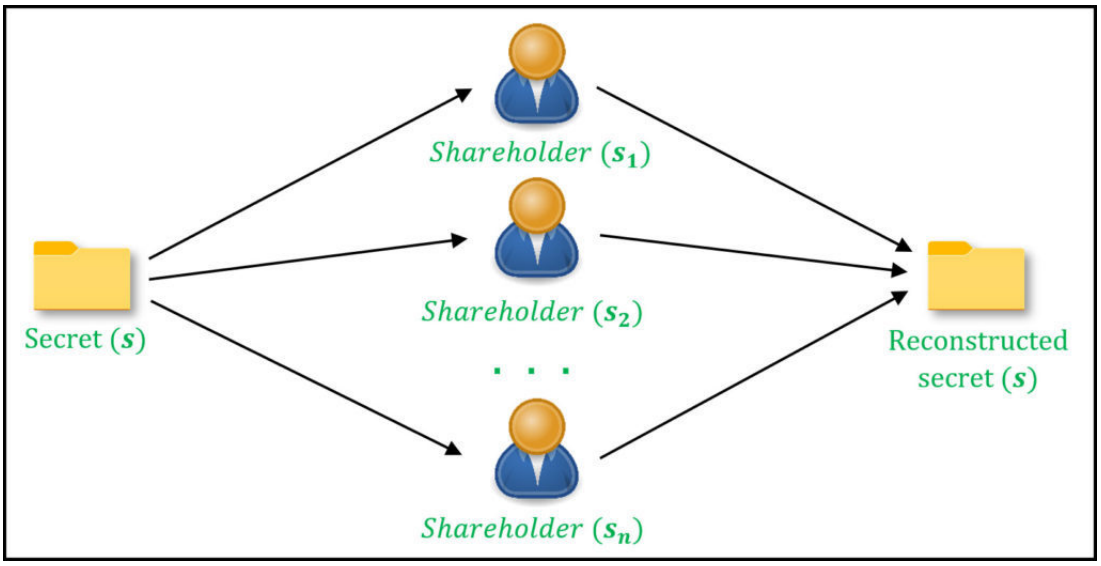


Figure 2.1: A simple secret sharing example.

homomorphic property, each party can use the shares it has from the previous wire to calculate the shares of the next computation without unsharing any value, thereby keeping the original secret values secure.

Each of the k parties receives a single share of each value in the MPC protocols used, and all of the shares are required for reconstruction. As a result, the sharing schemes can be classified as *k-out-of-k* sharing schemes. As such, the following schemes were considered:

2.1.1 Additive Secret Sharing

Additive secret sharing is a simple secret sharing scheme which entails creating k shares which, when added together, recreate the original shared secret. It is usually used within a finite field.

A simple way of creating such a scheme with k shares is by generating *k-1* random numbers within a finite field which will be used as the first *k-1* shares, and then calculating the last share $S(k) = V - \sum_{i=1}^{k-1} S(i)$ where V is the value to be secretly shared.

This way the original value can be reconstructed simply by adding all the shares together as $V = \sum_{i=1}^{k} S(i)$.

2.1.2 Shamir's Secret Sharing

Shamir's secret sharing is another homomorphic secret sharing scheme that uses finite field arithmetic which, unlike additive secret sharing, allows for reconstruction of the secret with *k-out-of-n* shares, meaning that a threshold k can be defined as a minimum of shares needed for recovery of the secret.

Just like as additive secret sharing *k-1* random numbers are generated. Then, using the secret and the generated random numbers, a secret share producing polynomial function is created. For example, with *n=k=4*, the function $f(x) = S + k0x + k1x^2 + k2x^3$ where S is the secret and k1 to k3 are the generated random numbers would be built.

Each share is a pair *(x,f(x))* and can be created after the function is built. The original secret can be reconstructed by calculating Lagrange basis polynomials with k shares and rebuilding the original polynomial function, which reveals the secret.

2.1.3 MPC protocol specific secret sharing

The MPC protocols that have been implemented use specific secret sharing schemes that can be adapted from those described above. They can be described as additive sharing schemes, but they take some extra steps to meet the protocol's requirements. These will be discussed in detail in the section that follows, when each respective protocol is detailed.

2.2 Secure multi-party computation

The main goal of a secure multi-party computation protocol (MPC) is to allow various participants to safely collaborate in executing a circuit of functions or wires over potentially secret data, while still reaching a common output. Usually, this circuit and all operations within it are known to all participants, including which parties must provide which inputs. These inputs however, must be kept secret throughout the cooperative execution.

Because most MPC protocols require the use of a secret sharing scheme to maintain the confidentiality of the secret inputs, these protocols can be divided into three parts. To begin, each secret input is shared by its respective party during the protocol's initialization or input sharing phase. Each computation is calculated in order after each party has the necessary data to execute the circuit. Depending on the protocol and calculation, this could be done asynchronously or in parallel amongst all parties. In most MPC protocols, certain operations, such as multiplication, must be calculated jointly by all parties, as will be explained shortly. In the final unsharing phase, once each party has reached a final output, this output can be publicly shared among all parties so that the required shares can be joined together to reveal the final output value.

As previously stated, not all operations can always be calculated on a shared value by each party without cooperation from others. For example, in most cases of a multiplication in a given MPC protocol this is challenging to do. Let's say there exists shared values x and y where $x_1, x_2, ..., x_k$ and $y_1, y_2, ..., y_k$ represent their k shares, and each party has one of them. The multiplication of x^*y in an additive sharing scheme can also be described as $(x_1 + x_2 + ... + x_k) * (y_1 + y_2, ... + y_k)$ which in turn means that to get the final value, each multiplication pair of $x_1y_1 + x_1y_2 + x_2y_1 + (...) + x_ky_k$ needs to be calculated. However, if each party only has one share, certain multiplication pairs cannot be calculated (such as x_2y_1 or x_1y_2),

necessitating the parties' cooperation in this computation and sharing data in such a way that no secrets are compromised.

On another hand, with the same example case of sharing scheme, an addition gate could be calculated in parallel by each party as $x+y$ and $(x_1 + x_2 + ... + x_k) + (y_1 + y_2, ... + y_k)$ can be described as $x_1 + x_2 + ... + x_k + y_1 + y_2, ... + y_k$ whose components can be calculated separately by each party (In this case each one would calculate $x_p + y_p$ where p is the index of the party).

Because this work is focused on the implementation of already detailed protocols, the security scope of these will not be extensively explored. Nonetheless, these protocols assume an honest majority were in k parties, any set of attackers of size $(k-1)/2$ can't reveal any secret inputs.

The sections that follow will go into greater detail about some of the protocols that were used in the implementation.

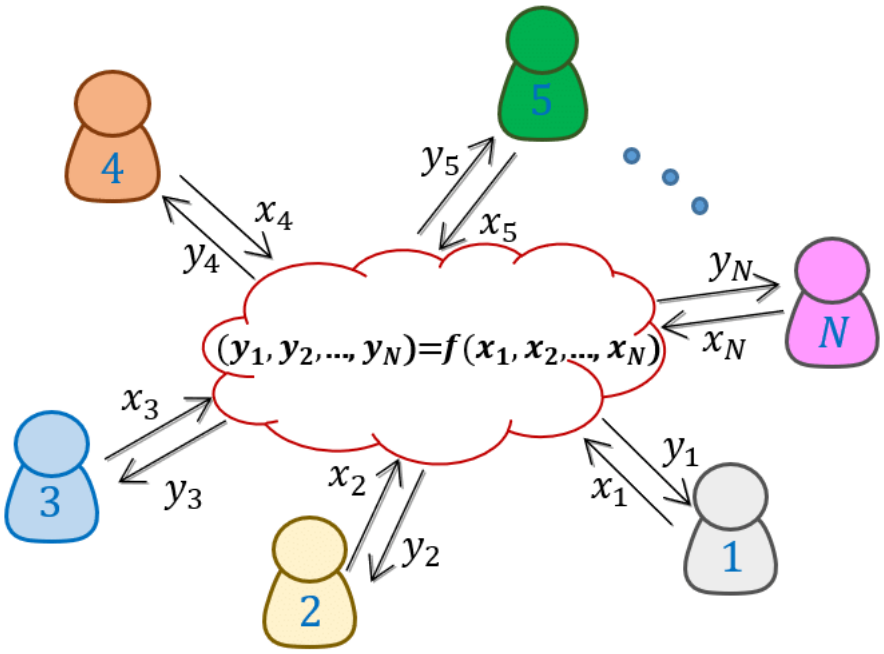


Figure 2.2: Secure Multi-Party Computation

2.2.1 Maurer's MPC

In Ueli Maurer's Secure multi-party computation made simple[18], Maurer describes an MPC protocol that is passively secure and requires a *k-out-of-k* sharing scheme. Because it is a passively secure protocol, for N parties, the shares are distributed in such a way that if $N/2$ parties conspire together and reveal their secret shares to each other, no secrets are revealed. The following steps are taken to create this sharing scheme for an arbitrary number of parties:

- Let $\sum = \{T_1, T_2, .., T_k\}$ in which each T_i is one of the maximal sets of possible adversaries, with each composed of $N/2$ parties. The total number of these sets is k.
- Each secret is divided into k secret shares.
- Each member of $\overline{T_i}$, which is composed of the parties not found in each T_i, secretly receives the share i of the k shares.

Because all of the shares are required to reconstruct the secret in this sharing scheme, only a number of parties greater than $N/2$ are capable of breaching security, making this scheme secure against passive corruption[18].

The number of allowed adversaries, calculated from the total number N of participating parties, determines how many and how the shares will be distributed. As a result, specific examples of this sharing scheme will be provided in the chapter on Development.

Using this sharing method and an additive sharing scheme, calculating linear functions such as an addition can be done by each party locally without communication between parties, simply by adding shares together[18]. On the other hand, multiplication can be calculated by doing the following protocol:

- Before the MPC is executed, for each party, the possible multiplication pairs that can be calculated are computed and chosen:
  - Let M be a matrix of sets in which each M_{ij} is the intersection $\overline{T_i} \cap \overline{T_j}$ (where T*'s* are the same sets used in the sharing scheme).
  - Each party of each set M_{ij} can calculate the multiplication pair $x_i y_j$ where x_i is the share i of the secret value x and y_j is the share j of the secret value y of any previously shared values x and y for the multiplication x*y.
  - For each party, choose which multiplication pairs it will calculate from those it can compute in such a way that there is no overlap between parties and each pair is calculated by one party.
- Each party calculates the multiplication pairs it was chosen to calculate, adds them together and secretly shares this sum between all parties using the sharing scheme.
- Each party receives N messages containing shares as described in the sharing scheme. The shares of same index of each message are added together and the final shared values of that multiplication are obtained by the parties.

This protocol's security is dependent on the security of both the communication channel used between the parties and the sharing scheme. To summarize, the multiplication protocol works by assigning specific multiplication pairs to each party to calculate, adding these pairs together, having the parties secretly share the obtained values with each other, and finally having each party add up all of the shares they received.

In a practical setting, the most complicated part of this protocol is assigning the multiplication pairs between parties, which is dependent on the total number of parties. As with the sharing scheme, the Development chapter describes a more practical approach to this, including examples for specific total number of parties and the challenges associated with assigning the multiplication pairs with this protocol.

2.2.2 Goldreich-Micali-Wigderson Protocol

A similar but simpler protocol known as the Goldreich, Micali, and Wigderson (GMW) protocol can be used for a multi-party circuit composed of boolean functions[12]. This protocol can be used for both arithmetic and boolean circuits work. The protocol's description that this work focused on can be found in David Evans, Vladimir Kolesnikov, and Mike Rosulek's *"A pragmatic introduction to secure multi-party computation."*[16].

In this protocol, the secret sharing of the inputs works over input bit by input bit. As a result, in this work, each input was considered a bit, and the circuits were represented as boolean functions. In this manner, the secret sharing is done in two steps as follows:

- For each secret bit of a given party, a random bit or bit mask is generated and distributed to each other party. This will be their share for the input gate.
- The share for the party with the secret is obtained by XORing each random bit mask shared and the secret bit together.

As a result, each party has a single share represented by a bit for each gate. The two main Boolean operations that are described in the protocol are the AND and XOR gates.

Much like the addition gate in the Maurer protocol, the XOR gate can be done locally without communication if each party XOR's the respective gate's shares. E.g. the local party's share of a XOR gate obtained from the result of gates with index i and j can be obtained by XORing the local share of the respective gates.

In an AND gate the same logic for the multiplication gates described before applies. If a gate describes $c = a \wedge b$ with the value's shares being $a_1, ..., a_n$ and $b_1, ..., b_n$ where a_1 is party 1's sharing of a, a_2 is party 2's sharing of a and so on and so forth, c can be rewritten as $c = (a_1 \oplus ... \oplus a_n) \wedge (b_1 \oplus ... \oplus b_n)$. Similarly to obtain the final values, the parties need to compute all of the pairs of $a_i \wedge b_j$. These pairs are jointly calculated as follows:

- Shares that can be used to obtain $a_i \wedge b_i$ are owned by a single party, and as such are calculated locally;
- To obtain $a_i \wedge b_j$ where $i \neq j$ means that two parties P_i and P_j need to communicate and calculate the pair together. To do this, they follow a communication based on the secret sharing algorithm:

  - P_i generates a random bit mask r and XOR's this value with a_i obtaining a masked value $m = a_i \oplus r$;
  - P_i sends the masked values r and m to P_j;
  - P_j chooses whether to keep r or m depending on the value of b_j: if b_j is 0, r is kept, if it's 1, m is kept as P_j's sharing for the pair $a_i \wedge b_j$;
  - P_i keeps r as it's share for the pair $a_i \wedge b_j$.

- Each party XOR's all of the shared values obtained and gets their final shared value of the gate.

The communication done in this algorithm is done in such a way that P_j doesn't learn the value of a_i and P_i doesn't learn b_j, keeping the security intact. This type of communication is called Oblivious transfer and will be further explained in section 2.2.3. In sum, to obtain their share in an AND gate each party P_i will be XOR'ing together the following values:

- The value obtained from $a_i \wedge b_i$;
- The r's where P_i acted as a sender in the communication;
- The messages chosen where P_i acted as a receiver.

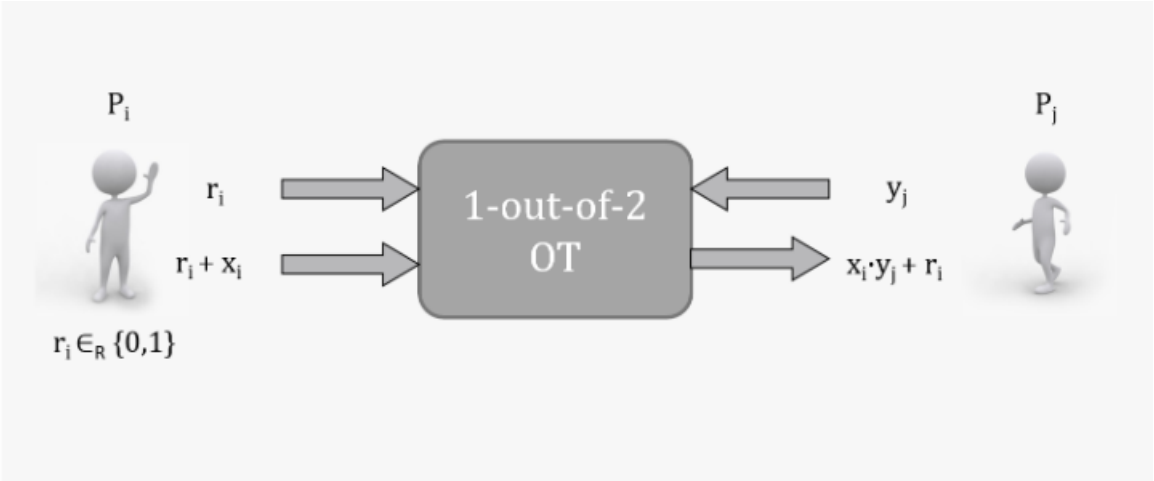


Figure 2.3: The 1 out of 2 oblivious transfer used in GMW

2.2.3 Oblivious Transfer

The Oblivious transfer communication protocol, as previously stated, is a required component in the AND operation of the GMW multi-party computation protocol.[16]

The main purpose of the Oblivious transfer is to serve as a communication channel between two parties, where the sender transmits many pieces of information to the receiver, but the receiver is only able to receive one of these, while the sender is unaware or oblivious of which piece of information was transferred.[20]

The type of Oblivious transfer used in the GMW protocol is called 1 out of 2 Oblivious transfer in which the sender transmits two messages *m0* and *m1* and the receiver uses a secret value y_j to determine which message they will receive. The protocol must also ensure that the sender doesn't have a way to learn what y_j is while still ensuring that the receiver can only obtain one of the messages.

While this work did not create an oblivious transfer implementation from the ground up like the MPC protocols, it is still important to have an idea how they ensure security as they are a building block for many MPC protocols. One method of implementing this protocol, for example, is to modify the Diffie-Hellman key-exchange protocol[13].

2.3 The Jasmin language

As mentioned previously, the protocol was developed using Jasmin. This language was chosen because it was designed for writing efficient cryptographic code while smoothly combining high-level and low-level constructs, allowing for control over low-level programming details that might be critical for performance while retaining programming logic used in higher level languages. It has also been used in multiple cryptography related implementations[2][3].

While it is a low-level programming language with access to assembly level instructions on data registers and stack, it also has high-level structured control flow features such as loops and procedure calls. The jasmin compiler converts code into an assembly program, which can then be called in more commonly used languages such as C, making integration with other applications easy.[1]

Jasmin functions are divided into two categories: export functions, which can be called and integrated into other programs after the jasmin code has been compiled, and inline functions, which can only be called from within the jasmin code. Similarly, variables must be declared as register variables if they are to be kept in register memory, stack variables if they are to be kept in the stack, or inline variables, which will be replaced by fixed constant values at compilation time.

To facilitate compilation and debugging, Jasmin code should be written in such a way that each line of code translates into one assembly command and that values stored in register memory can be used (eg: using more than the amount of available register addresses at one give time, not having a free register address for the division operation, etc.). While the jasmin compiler can handle some of these situations, namely freeing up register values when they are no longer needed, there may be code that it cannot convert into an assembly program, in which case errors

will be thrown during compilation.

```
//XORs values in pointers contained in v1S and v2S, puts in resS.
inline
fn XOR_wrapper(stack u64 v1S v2S resS){
   reg u64 v1;
   reg u64 v2;
   reg u64 v3;
   reg u64 aux;


   aux = v1S;
   v1 = [aux];

   aux = v2S;
   v2 = [aux];

   v3 = v1 ^ v2;
   aux = resS;
   [aux] = v3;
}
```

Code Block 2.1: Example of jasmin code.

As an example, Code Block 2.1 shows a inline function that receives three pointers contained in stack variables, retrieves the values contained in the first two pointers, XOR's them together, and puts the result in the pointer contained in the last stack input variable. All other variables use register addresses.

The jasmin language also has a number of verification tools that are useful for security-focused programming and implementations. At compile time, the safety checker tool can be used as a flag to instruct the jasmin compiler to check export functions for valid termination, correct memory use (arrays are in bounds, memory accesses are properly aligned and target allocated memory), and to check if arithmetic operations are applied to the valid arguments. To mathematically ensure the compiler maintains cryptographic properties found in the code the compiler allows to extract Jasmin program to an equivalent EasyCrypt[24] model, which is a tool-set that can be used for verification of cryptographic proofs. These are further described in the jasmin language GitHub repository[23] .

Coq, a formal proof management system, was used to fully define and verify the jasmin language and jasmin to assembly compiler. This means that the generated assembly code is predictable and always correct in relation to its jasmin source. With the jasmin code extraction to the EasyCrypt model described above, this allows for the creation of easily-verifiable assembly language code while still retaining access to high level code features. Because of these characteristics, the jasmin language was chosen for this work.

However, as jasmin is a newer language, one of the main challenges is the lack of cryptographic

features and libraries that are usually already implemented in other languages. Another potential difficulty with developing in Jasmin stems from its design as a low-level language. As such, the programmer must be aware of how they are manipulating memory registers and stack.

Because handling complex operations, such as communication between parties, would be difficult for jasmin code the libraries written in jasmin handle the key operations of the MPC protocols, such as secret sharing, message creation and processing, and secret rebuilding via unsharing.

Chapter 3

State of the Art

In order to create a more useful MPC implementation with more applications it is necessary to learn and document the current use of the technology. While MPC is still a relatively new and active area of research, this chapter serves to summarize successfully deployed MPC solutions and implementations.

Practical deployment of MPC systems are few and far between, not only due to the freshness of the technology, but also for other reasons that stem from it. Most new security technologies suffer from a well-founded reluctance to trust the underlying science and first-time implementations that may contain unforeseen but easily exploitable vulnerabilities in a practical setting. When a decision maker or system designer is in charge of a system's security, they may be hesitant to use MPC, an issue that will only be resolved with further exploration and documentation of MPC practical applications[16]. As a result, the following section describes some of these.

3.1 MPC Deployments

The first large scale deployment was found in the January 2008's Danish sugar beets auction. This auction came to be from 5000 Danish farmers who grow sugar beets and have contracts with Danisco, Denmark's sole major sugar producer. Danisco had to close one of their factories due to the EU drastically reducing subsidies, resulting in a nationwide need for a market for these farmers. While the farmers did not want to rely on Danisco to protect their financial information during the bidding process, Danisco also wanted to be involved in all of the bidding as their contracts could reveal sensitive inside and otherwise secure company information to a third party. As a result, they collaborated with a group of researchers (SIMAP project) to deploy a large decentralized auction that employs a three-party MPC as the auctioneer, ensuring the privacy of both the Danish farmers and Danisco.[11]

Another large-scale government deployment of MPC occurred in Estonia. In 2012, nearly 43% of IT students enrolled in the previous five years had failed to graduate, an unusual occurrence in a country with advanced e-government and technology awareness. As a result,

the Estonian Association of Information and Communication Technology wanted to investigate, but due to information security and privacy laws, the Estonian Association of Information and Communication Technology was unable to easily obtain and process student private information, such as tax records, within the Ministry of Education and the Tax Board. As a result, they used a three-way MPC with the help of the company Cybernetica's Sharemind framework[9] to securely collect and handle student private information. They discovered that there was no link between working while studying and failing to graduate on time, but that more education was associated with higher income.[10]

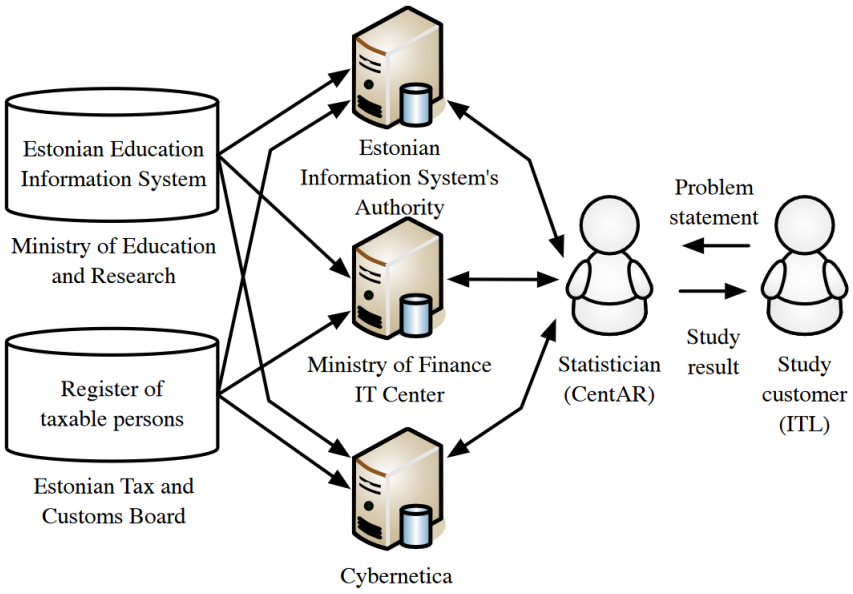


Figure 3.1: The Estonian Student study entities.

A similar situation occurred in Boston, where the Boston Women's Workforce Council (BWWC) sought to identify salary disparities among different employee gender and ethnic demographics at various levels of employment. However, this survey would need to handle sensitive and private information, such as salary data, as in Estonia's case. In response to this demand, Boston University researchers created and developed a web-based MPC aggregation tool that allowed companies to submit wage data anonymously and with full technological and legal protection for the study's aims. The inclusion of secure multi-party computation as a condition in a bill for student data analysis recently filed in the United States Senate is one indirect effect of this effort.[8]

A different application of MPC technology can be found in Unbound Tech's[25] key management products. Essentially, this application of MPC adds an extra layer of security by dividing critical keys into two or more shares, which are then stored on different servers, rather than keeping the keys in a single location. To use these keys, an MPC operation would be performed

between the authentication server and the key-holders. This means that even if an attacker gains access to one of these secure locations, they will be unable to use the key for malicious purposes.

3.2 MPC Implementations

While MPC is a relatively new technology, some implementations and frameworks are already available to the public. This section will go over a few examples.

An example of such implementations is ABY, which aimed to combine secure computation schemes based on Arithmetic sharing, Boolean sharing, and Yao's garbled circuits and that makes available best practice solutions in secure two-party computation. [14] While ABY is described as being focused on efficiency, it is limited to two-party joint computation, and the code available is described as being for testing purposes, which cannot guarantee security and correctness.

A protocol framework based on the same base technologies as ABY named ABY3 also exists, developed with the aim of performing machine learning algorithms with training and evaluation of models on joint data using three-party computation.[19]

HoneyBadgerMPC is a MPC-based confidentiality layer for blockchains focused on robustness. This use of MPC isn't aimed on techniques such as zero knowledge proofs, but on smart contracts and blockchain integration.[21]

Fancy-garbling allows for Boolean and arithmetic circuits and is currently incorporated into a group of secure MPC focused libraries called swanky. It was originally designed for two-party computation using garbled circuits, and it is now implemented in Rust.[6]

Obliv-C is a GCC wrapper with the aim of embedding secure computation protocols inside regular C programs.[27] It allows for use of specific operations that will then be used to call the Obliv-C MPC code. It allows for two party computation developed with garbled circuits, and is secure against semi-honest adversaries.

These implementations of MPC discussed so far fall outside of the scope of this work, which is an implementation of secure full multi-party computation aimed at being integrated into various purposes, namely Zero Knowledge, that is efficient and most importantly verifiable as secure. Furthermore, the majority of these use Yao's garbled circuit protocol, with some also employing the GMW protocol. Maurer's Protocol is not used in any of the deployments mentioned.

The EMP-toolkit allows for two party and multi party computation, with implementations secure against semi-honest adversaries and others resistant against malicious parties.[26] The final protocol used by EMP is presented as efficient and with scalability to allow a large amount of parties. While this implementation overlaps with much of this work, the MPC protocol used by the toolkit was created by extending a two party computation protocol. This work aims to create an implementation based on multi party protocols from the ground up. A comparison of a implementation developed in this work and highly optimized implementations used in a zero

knowledge setting such as the toolkit can be found in Machine-checked ZKP for NP-relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head[4].

3.3 Implementations in Jasmin

While jasmin is a new language, it has already been proven to be effective in the development of cryptographic protocol implementations.

One of the major projects implemented in jasmin was a efficient SHA-3 hash function with proved correctness and security and with side-channel protection[2]. In the same vein, the implementation of ChaCha20-Poly1305 in jasmin is as an example of the approach for building cryptographic implementations in efficient and verified assembly code using jasmin and EasyCrypt described in the last chapter[3].

Other libraries and frameworks have also been developed in jasmin. In the scope of this work, a useful library available implements operations in a finite field used for zero knowledge proofs referred to as zK field (defined by the prime number $2^{255} - 19$). This library will be used for the MPC implementation designed for use in zero knowledge proofs.

Chapter 4

Maurer's MPC Development

One of the primary goals of this work was to address, document, and resolve the issues that arise during the implementation of an MPC protocol in an efficient, secure, and practical manner. As a result, the sections that follow describe these issues and the solutions that were discovered to resolve them. Aside from that, the reasoning behind design decisions and the process that led to the final implementation will also be explained.

To create the most efficient program possible, it was decided that the final implementation would have to focus on a specific number of parties, and the final programs are aimed at MPC for three or five participants. The process described in the following sections however, can be applied to a different number of parties, and the final algorithms can be easily adjusted without causing any changes to their logic.

4.1 Prelemenary Prototyping

So as to better understand how to go about implementing a MPC protocol, a prototype of Maurer's passively secure MPC was created. This protocol was chosen as the basis for the prototype because it was simple while also being generic enough to be able to execute many different types of circuits and reveal any issues that may arise as a result.

Python was chosen as the language for this prototype because the sharing scheme and multiplication part of the protocol use operations over sets as a base. An example of these operations would be Python's list comprehension which was useful in quickly and easily implementing these components.

```
def __getTcomplement(Nparty):
    Parties = [i for i in range(Nparty)]
    T = combinations(Parties,Nparty/2)
    Tc = [ [p for p in Parties if T[i].count(p) == 0] for i in range(len(T))]
    return Tc
```

```
def __getM(Tc):
    return [[[m for m in __intersect(Tc[i],Tc[j])] for j in range(len(Tc))] for
        i in range(len(Tc))]
```

Code Block 4.1: Creation of sets such as $\overline{T}$ and matrix M as described in the protocol.

4.1.1 Allocating multiplication pairs

The main challenge in prototyping the protocol was in how to choose which parties calculate specific multiplication pairs in the multiplication component of the protocol.

To address this, a program was developed that calculated which pairs each party could perform with an arbitrary number N of parties. In this program, a matrix U with N rows is created, each containing the set of pairs that each party is capable of calculating. This U is created by using the matrix M described in the multiplication protocol. The following image 4.1 is the matrix obtained from 5 participating parties:

```
U (Full):
[(4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (6, 9),
 (7, 4), (7, 5), (7, 6), (7, 7), (7, 8), (7, 9), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9)]
[(1, 1), (1, 2), (1, 3), (1, 7), (1, 8), (1, 9), (2, 1), (2, 2), (2, 3), (2, 7), (2, 8), (2, 9), (3, 1), (3, 2), (3, 3), (3, 7), (3, 8), (3, 9),
 (7, 1), (7, 2), (7, 3), (7, 7), (7, 8), (7, 9), (8, 1), (8, 2), (8, 3), (8, 7), (8, 8), (8, 9), (9, 1), (9, 2), (9, 3), (9, 7), (9, 8), (9, 9)]
[(0, 0), (0, 2), (0, 3), (0, 5), (0, 6), (0, 9), (2, 0), (2, 2), (2, 3), (2, 5), (2, 6), (2, 9), (3, 0), (3, 2), (3, 3), (3, 5), (3, 6), (3, 9),
 (5, 0), (5, 2), (5, 3), (5, 5), (5, 6), (5, 9), (6, 0), (6, 2), (6, 3), (6, 5), (6, 6), (6, 9), (9, 0), (9, 2), (9, 3), (9, 5), (9, 6), (9, 9)]
[(0, 0), (0, 1), (0, 3), (0, 4), (0, 6), (0, 8), (1, 0), (1, 1), (1, 3), (1, 4), (1, 6), (1, 8), (3, 0), (3, 1), (3, 3), (3, 4), (3, 6), (3, 8),
 (4, 0), (4, 1), (4, 3), (4, 4), (4, 6), (4, 8), (6, 0), (6, 1), (6, 3), (6, 4), (6, 6), (6, 8), (8, 0), (8, 1), (8, 3), (8, 4), (8, 6), (8, 8)]
[(0, 0), (0, 1), (0, 2), (0, 4), (0, 5), (0, 7), (1, 0), (1, 1), (1, 2), (1, 4), (1, 5), (1, 7), (2, 0), (2, 1), (2, 2), (2, 4), (2, 5), (2, 7),
 (4, 0), (4, 1), (4, 2), (4, 4), (4, 5), (4, 7), (5, 0), (5, 1), (5, 2), (5, 4), (5, 5), (5, 7), (7, 0), (7, 1), (7, 2), (7, 4), (7, 5), (7, 7)]
```

Figure 4.1: The matrix U for 5 parties.

With 5 parties the number of sets in T is 10, so each value will have 10 shares, ordered from 0 to 9 in this example. As described in the image 4.1. The first party can obtain the pairs $x_4y_4, x_4y_5, x_4y_6, x_4y_7, x_4y_8, x_4y_9, x_5y_4, (...)$. Likewise, both the 3rd, 4th and 5th party can calculate the pair x_0y_0.

Despite the fact that this U allows for manual selection of which pairs are computed by which parties, there still is overlap. This overlap means that if this matrix were used as is, the parties would have to synchronize which pairs they would make before running the circuit. This is due to the fact that each pair must only be added to the final result once between all parties.

To remove this costly initialization, an alternate and simple method of calculating U was created. This method uses a greedy heuristic for associating pairs with the parties. For each pair, the first party that can compute it is chosen. This ensures that there is no overlap between parties.

The issue with this approach is that each party computes a different number of pairs, as

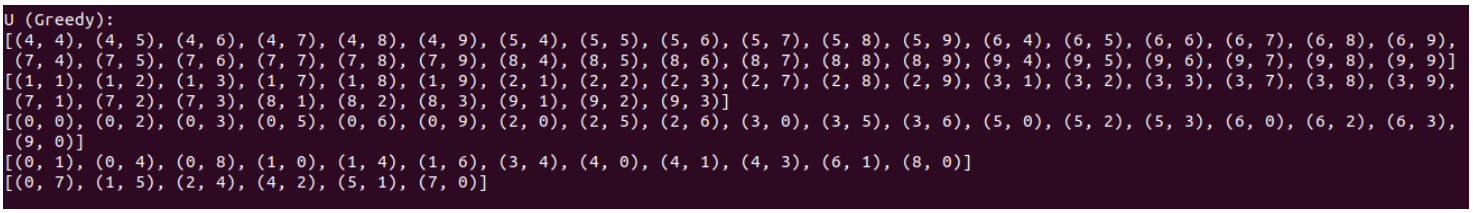


Figure 4.2: The matrix U for 5 parties with a greedy heuristic.

shown in the example image. To counteract this, another algorithm based on the greedy one was developed.

Assuming the intersection matrix M is created from $\overline{T_i} \cap \overline{T_j}$ as described in the protocol, where T's sets are ordered in ascending order. Let there be a value pi that ranges from 0 and the length of each set inside M, and value $maxP$ which is the number of pairs each party will do in a balanced scenario ($maxP = Total\_number\_of\_pairs/N$). Starting from the last pair in the matrix:

- Add 1 to pi in it's range, looping it back to 0 if it currently is it's max value.
- In the set of parties allowed to do the current pair, check if the party of index pi has already been allocated $maxP$ pairs.
  - If not, choose this party to do this pair and exit the loop of this pair.
- If the party already has the max number of pairs, repeat the last steps until a party is found.
- If no party has free spots for a pair, add the pair to one of them randomly.
- Proceed the next pair in reverse order, keeping the last pi value, and repeat this loop until all pairs are allocated.

This algorithm's logic is also greedy, but tries to balance out the pairs between all parties. While it may not work with all N number of parties, due to how M is constructed, if a T is created with it's sets in ascending order, it will work for the N's tested. Either way, this is a better approach for creating the matrix U than using a blind greedy algorithm.

As seen in figure 4.3, each party computes the same amount of pairs, has no overlap, and every pair out of the 100 is allocated.

4.1.2 The ideal multiplication pair allocation

To be as efficient as possible however, it would be ideal for each party to run the same code for multiplication, and not depend on checking which pairs it would have to compute. This ideal

```
U (Greedy Balanced):
[(9, 4), (8, 5), (7, 6), (6, 7), (5, 8), (4, 9), (9, 7), (9, 5), (8, 7), (8, 4), (7, 8), (7, 4), (6, 8), (6, 4), (5, 7), (5, 4), (4, 7), (4, 5),
 (7, 7), (5, 5)]
[(9, 1), (8, 2), (7, 3), (3, 7), (2, 8), (1, 9), (9, 8), (9, 2), (8, 9), (8, 1), (7, 9), (7, 1), (3, 8), (3, 2), (2, 7), (2, 3), (1, 7), (1, 3),
 (9, 9), (3, 3)]
[(9, 0), (6, 2), (5, 3), (3, 5), (2, 6), (0, 9), (9, 6), (9, 3), (6, 9), (6, 5), (6, 0), (5, 9), (5, 6), (5, 0), (3, 9), (3, 0), (2, 9), (2, 0),
 (0, 5), (0, 3)]
[(8, 0), (6, 1), (4, 3), (3, 4), (1, 6), (0, 8), (8, 6), (8, 3), (6, 3), (4, 8), (4, 6), (4, 0), (3, 6), (3, 1), (1, 8), (1, 0), (0, 6), (0, 1),
 (8, 8), (6, 6)]
[(7, 0), (5, 1), (4, 2), (2, 4), (1, 5), (0, 7), (7, 5), (7, 2), (5, 2), (4, 1), (2, 5), (2, 1), (1, 4), (1, 2), (0, 4), (0, 2), (4, 4), (2, 2),
 (1, 1), (0, 0)]
```

Figure 4.3: Balanced U for 5 parties found with balanced greedy heuristic.

scenario could be realised if each multiplication pair represented the same share indexes for each party.

For instance, using an example of three parties adhering to the protocol, each value is divided into 3 shares ($\overline{\sum} = 3$), each party has 2 shares ($\overline{T_i} = 2$) and the sharing scheme is as shown in table 4.1.

	Share 0	Share 1
Party 1	1	2
Party 2	2	0
Party 3	0	1

Table 4.1: Sharing scheme for 3 parties

In this case, it would be ideal if each party's first multiplication pair was, say, x_2y_1, x_0y_2 and x_1y_0 respectively because each party would multiply the shares with indexes (1,0), allowing each party, regardless of who they are, to run the same code for this pair (I.e., multiply the share located in index 1 with the one located in index 0). If all of the parties' pairs followed this pattern, the code for the multiplication operation would be the same for all of them, eliminating the need for conditional statements and increasing efficiency.

This would mean that a perfect multiplication pair allocation would be one where all parties compute the same number of multiplication pairs, there is no overlap of pairs between each one, and all pairs represent the same share indexes for each party. Because, as mentioned before, the final implementation would be aimed at three and five parties, this allocation would be needed to be found for these two situations.

A brute-force search was conducted using these two sharing schemes and the respective full matrix U of possible multiplication pairs per party in order to find an ideal multiplication pair allocation for both of these situations.

The multiplication pair allocation described in 4.2 was discovered for three parties. As can be seen, each party follows the pattern of computing the values in the sharing scheme's indexes (0,0), (0,1), and (1,0). I.e.: The shares represented by indexes (0,0) in the sharing scheme are $x1y1$, $x2y2$ and $x0y0$ for each party respectively, so this is the first of the 3 multiplication pairs they calculate.

	Pair 1	Pair 2	Pair 3
Party 1	x_1y_1	x_1y_2	x_2y_1
Party 2	x_2y_2	x_2y_0	x_0y_2
Party 3	x_0y_0	x_0y_1	x_1y_0

Table 4.2: Ideal multiplication pair allocation with 3 parties.

With this allocation, there is no overlap between all parties, all nine multiplication pairs of $(x_0 + x_1 + x_2) * (y_0 + y_1 + y_2)$ are calculated and the same code can be used between all parties using the respective sharing scheme indexes (0,0),(0,1),(1,0).

The same process was done with five participating parties. In this case, when adhering to the protocol, each value is divided into 10 shares ($\overline{\sum} = 10$), each party has 6 shares ($\overline{T_i} = 6$) and the sharing scheme is as represented in table 4.3.

	Share 0	Share 1	Share 2	Share 3	Share 4	Share 5
Party 1	5	9	8	7	4	6
Party 2	1	2	9	3	8	7
Party 3	6	0	2	5	9	3
Party 4	3	8	4	6	0	1
Party 5	7	4	0	1	2	5

Table 4.3: Sharing scheme for 5 parties

Because there are many more multiplication pairs per party (from 3 to 20), calculating an ideal multiplication allocation table for 5 parties was much more difficult than for 3. Despite this, the allocation was found and is described in tables 4.4 and 4.5.

	Pair1	Pair2	Pair3	Pair4	Pair5	Pair6	Pair7	Pair8	Pair9	Pair10
P1	x_5y_5	x_5y_9	x_5y_8	x_5y_4	x_9y_7	x_9y_4	x_9y_6	x_8y_5	x_8y_8	x_8y_7
P2	x_1y_1	x_1y_2	x_1y_9	x_1y_8	x_2y_3	x_2y_7	x_2y_8	x_9y_1	x_9y_9	x_9y_3
P3	x_6y_6	x_6y_0	x_6y_2	x_6y_9	x_0y_5	x_0y_3	x_0y_9	x_2y_6	x_2y_2	x_2y_5
P4	x_3y_3	x_3y_8	x_3y_4	x_3y_0	x_8y_6	x_8y_1	x_8y_0	x_4y_3	x_4y_4	x_4y_6
P5	x_7y_7	x_7y_4	x_7y_0	x_7y_2	x_4y_1	x_4y_2	x_4y_5	x_0y_7	x_0y_0	x_0y_1

Table 4.4: First 10 pairs of the ideal multiplication pair allocation with 5 parties.

While brute force searching worked with five parties, it is likely to fail with a larger number of parties. The algorithm used to find the optimal allocation had to trim the search to the U tables of 5 parties and took several hours to complete. As a result, finding an optimal allocation for a larger number of parties would necessitate a different solution.

For these allocations, the indexes that will be used in the code to select each party's pairs are for 3 parties, as mentioned before, (0,0), (0,1), (1,0), while for 5 parties it's (0,0), (0,1), (0,2), (0,4), (1,3), (1,4), (1,5), (2,0), (2,2), (2,3), (2,4), (3,0), (3,1), (3,2), (3,5), (4,1), (4,2), (4,3), (5,0), (5,3).

	Pair11	Pair12	Pair13	Pair14	Pair15	Pair16	Pair17	Pair18	Pair19	Pair20
P1	x_8y_4	x_7y_5	x_7y_9	x_7y_8	x_7y_6	x_4y_9	x_4y_8	x_4y_7	x_6y_5	x_6y_7
P2	x_9y_8	x_3y_1	x_3y_2	x_3y_9	x_3y_7	x_8y_2	x_8y_9	x_8y_3	x_7y_1	x_7y_3
P3	x_2y_9	x_5y_6	x_5y_0	x_5y_2	x_5y_3	x_9y_0	x_9y_2	x_9y_5	x_3y_6	x_3y_5
P4	x_4y_0	x_6y_3	x_6y_8	x_6y_4	x_6y_1	x_0y_8	x_0y_4	x_0y_6	x_1y_3	x_1y_6
P5	x_0y_2	x_1y_7	x_1y_4	x_1y_0	x_1y_5	x_2y_4	x_2y_0	x_2y_1	x_5y_7	x_5y_1

Table 4.5: Last 10 pairs of the ideal multiplication pair allocation with 5 parties.

4.1.3 Other challenges and design choices

While the multiplication pair allocation was the most difficult challenge at this step, other parts of the full system needed to be implemented in order to complete the full MPC protocol. The most important ones, as well as the design decisions that resulted from them, are described in this section.

4.1.3.1 Secret Sharing Method

As previously stated, secret sharing is one of the foundations of an MPC protocol, and the secret sharing schemes considered were a basic Additive Sharing method and Shamir's Secret Sharing due to their simplicity and homomorphic properties.

The method for sharing secrets however is easily interchangeable as long as it is homomorphic. In the prototype, this was tested by creating and calling these two different secret sharing methods from the same protocol code.

In the end, the basic Additive Sharing method was chosen because the MPC protocol does not require a k-out-of-n sharing structure (this protocol always requires all shares for a value to be rebuilt) and requires less computation to share and rebuild a secret.

4.1.3.2 Communication

While the communication methods used between parties are somewhat outside the scope of this work, as it is focused on the MPC protocol itself, it is still important to note that the communication systems used can affect the overall performance and security of the protocol's execution.

However, because of the way this protocol handles sharing, as long as half of the participants are not compromised, the protocol remains passively secure. This security can be enhanced by incorporating other commonly used security methods into the communications between parties, such as encryption for confidentiality, public key cryptography for authentication and non-repudiation, and so on.

While the communication channel might not be a big focus, the messages passed between parties are, but where not fully explored during this prototype. The messages created and received by the protocol are further detailed in section 4.2.

4.1.3.3 Inputs and outputs

While input sharing and the unsharing of the output are important parts of the protocol, Ueli Maurer's paper[18] does not formalize them.

Each participant can have multiple secret inputs and must share them between all parties following the sharing scheme before the circuit starts being computed.

As for the output, after all parties have completed the circuit, it was decided that they would all reveal and send their final shares to each other, and use these to unshare the final output. Because the sharing scheme is public and all parties know which participant each group of shares came from, the required shares can be chosen from each participant and the output rebuilt.

Each one of these steps where dubbed input and output gates in the design, much like an addition or multiplication computation is called an addition or multiplication gate respectively.

4.1.3.4 Public Inputs

While publicly known inputs have not been mentioned much until now, as they could simply be implemented and used as secret inputs, if the goal is to make an implementation as efficient as possible, secret sharing values known to all parties is redundant and counter-intuitive. These public inputs are also known as constant values or constants in the internal code and design.

However, using these values efficiently is more complicated than simply not sharing them and using them as-is in the computations. To begin, the gates are expecting shared values, which means that the constants must be shared locally. Because secrecy is not required, this is easily accomplished with additive shares without the need for randomness if the first share is value itself and all other shares are zero (I.e. if all the shares of a value v are unshared by being added up together, $v + 0 + ... + 0$ the value v is restored). Each party then selects it's respective shares (0 or v) and uses them in the rest of the computations. This operation was named a constant gate.

So as to be even more efficient, if a multiplication is between a publicly known value and a secretly shared one can be done locally. This is easily explained by multiplying a secret x with k shares and a public value p as $(x_0 + x_1 + ... + x_k) * p = px_0 + px_1 + ... + px_k$. As all px_i can be calculated between all parties without sharing values between them, these multiplications can be done locally. These operations were named simple multiplication gates or Smult gates.

4.2 Design

After having a working MPC prototype and understanding what components a practical implementation requires, the design for a MPC implementation that uses Jasmin was created. As previously stated, because Jasmin is a low-level language with limited access to code and libraries in comparison to more widely used languages, certain MPC components such as communication between parties would be extremely difficult to implement. Therefore, the code is separated into jasmin and a master program.

4.2.1 Circuit representation

To consistently depict and execute possible circuits, a simple representation of them had to be created. Following the python prototype and other requirements, it was decided that the following operations would be allowed in this design:

- Input gates, for each party's secrets;
- Constant or public input gates, for values known by all parties;
- Addition gates, for addition between two other gate values;
- Multiplication gates, for multiplication between two other gates.

The circuit is then represented by a set of gates, each of which is a set of values that describe it. For example, an input gate whose value is provided by the party with index 2 would be described as *(input,2)*, while *(addition,2,3)* would represent an addition gate that adds up the values of the gates with index 2 and 3. An important property of this representation is that the addition and multiplication gates require gates with previously set values and, as a result, cannot be the first gate in a given circuit.

An example of a representation of a small circuit with seven gates would be *{(constant,15), (input,0), (input,1), (input,2), (addition,0,1), (addition,2,3), (multiplication,4,5), (addition,6,1)}*. There are at least three parties in this circuit, and three of them use a secret input. Following the sharing of the secret input, the execution begins by adding the constant 15 and the secret input of party with index 0. The other two inputs are then added together, followed by multiplication of the values obtained from the previous two operations. Finally, 15 is added to the result of this multiplication to get the final output. This circuit is visually represented in figure 4.4.

Although they are not included in this design, output and simple multiplication gates are used in the implementation. The output is omitted because including it would be redundant due to the fact that all circuits in the implementation end with this gate. When a public value is

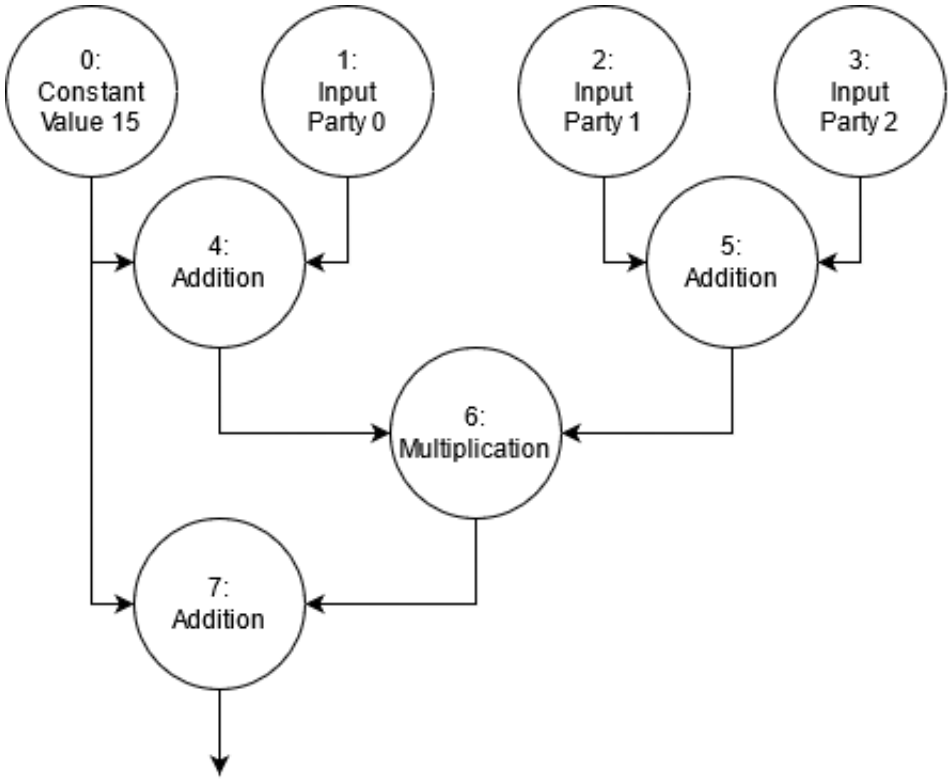


Figure 4.4: Visual representation of a circuit.

used in a multiplication, the simple multiplication gate is used and is omitted in this type of representation for the sake of simplicity.

4.2.2 Messaging

The protocol requires participants to exchange messages in the input, multiplication, and output gates in order for it to function. These messages contain shares of secret inputs or of values derived from them.

In an input gate, the party responsible for providing the secret creates messages containing the shares associated with each party and distributes them between them. The remaining parties wait for this message. With N participants, there will be one party sending 1 message to each other party, therefore a total of N messages are sent.

In a multiplication gate, after calculating it's respective sum of multiplication pairs, this value must be shared. As such all parties are have the responsibility of creating and sending messages containing the shares of these sums. With N participants, all parties must each create and send N messages, with a total of $N*N$ messages being sent between all parties. Similarly, during the output gate, all parties unshare their output, sending all of their results to all participants, resulting in the same amount of messages as in the multiplication operation.

For the unsharing at the end of the circuit, each party simply sends messages to the other

parties containing the final shared values obtained from the protocol's execution. With all the shares, each party can reconstruct the output and arrive at the same final value.

4.2.3 Jasmin calls

The jasmin portion of the implementation is limited to manipulating a memory space containing each party's current circuit values, also known as circuit status, and creating/processing messages. Because Jasmin code can be easily compiled to Assembly-language code, it was used as a framework that is called from a C program used to run the circuit.

In other words, whenever a party needs to run a gate or operation, it calls Jasmin, who then updates the relevant information in memory and/or creates and sends messages. Following information gathered from the prototype, the external calls deemed necessary to exist in order to compute as many circuits as possible in an efficient manner are:

- input_start: Called when starting an input gate. Creates the input messages to be distributed between players. Receives a secret input, the necessary random numbers to create shares and a space in memory to write the messages to send to all players;
- input_end: Called after having recieved the respective input's message from input start. Populates the party's status with the input's shares. Receives the space in memory containing the party's messages, the one containing the circuit status and the index of the current gate;
- const_start: Called when starting a public input gate. Creates the public input messages, from which the calling party will choose. Works similarly and receives the same values as input_start, but the messages are not sent outside of jasmin;
- const_end: Called right after const_start. Populates the party's status with the public input's shares. Receives the party's shares of the public input, the space in memory containing the circuit status and the index of the current gate;
- add: Called when computing a addition gate. Adds the two wires' values and populates the party's status. Receives the space in memory containing the circuit status, the index of each of the gates being added, and the index of the current gate;
- Smult: Called when computing a multiplication between a secret and a public value. Multiplies the two gates' values and populates the party's status. Receives the space in memory containing the circuit status, the index of the gate containing the secret value, followed by the public one and the index of the current gate;
- mult_start: Called when computing a multiplication gate. Computes the party's multiplication pairs of the respective gates, adds them together and secretly shares the result. Receives the space in memory containing the circuit status, the index of each of the gates

being multiplied, and the index of the current gate, the necessary random numbers to create shares and a space in memory to write the messages to send to all players;

- mult_end: Called after having all party's multiplication messages. Adds together the shares received to get the party's multiplication gate's shared result and populates the status. Receives the space in memory containing the party's messages, the one containing the circuit status and the index of the current gate.
- out_start: Called at the end of the execution of the circuit to start the unsharing. Receives the party's circuit status and a space to write messages to. Writes the messages needed for the unsharing in this space.
- out_end: Called after receiving the unsharing messages. Receives these messages and a space in memory to write the circuit output. Unshares the value from the messages and puts it in the space received.

Certain calls, such as const_start, may appear to be redundant, but their necessity will be explained in later sections, such as, in this case, section 4.3.2.

4.2.3.1 Party and Field Paramenters

Many of the jasmin code's code blocks refer to specific global parameters. These are fixed or constant values that are important for the protocol implementation and are used in the code of many functions. These are described as follows:

- nShares: Total number of shares, followed by a number for the total number of parties (eg: nShares5 for 5 parties, which is 10);
- nSharesPP: Number of shares each party owns per wire, followed by a number for the total number of parties (eg: nSharesPP5 for 5 parties, which is 6);
- shareSize: The number of values each share contains. Depends on the Finite Field being used and its implementation.

Another value that appears frequently in the code is 8, which is the size that an unsigned integer takes up in memory, but it is omitted from the list above because it has little to do with the protocol or field implementations.

4.2.4 Finite Fields

Because one of the main goals of this implementation is to be easily integrated into other projects (specifically zero-knowledge proofs[22]), it is critical that the design allows for large Finite Fields

used in other protocols. This means that the secret values, and thus the shared values, can be used in fields that require the use of multiple memory addresses to represent a single value.

To accommodate this, jasmin code that interacts directly with any value must instead use a wrapper that can be adapted or rewritten to work with values that describe the field being used. Furthermore, the rest of the implementation expects the type of data used by the secret and shared values to be abstract, and values stored in multiple memory addresses or with large word sizes can be used.

An example of this is a zk Finite Field, used in MPC-in-the-head for zero knowledge proofs, where each value is represented in four 64 bit addresses. To summarize, whether each value is described in one memory address or many, as in this Finite Field, the main jasmin code remains the same, with the only difference being the operations wrapper's code.

```
  //For each shared i in (add,x,y), find the memory addresses of the share of
      xi and yi, add them together and put them in the right memory space
for shareN = 0 to nSharesPP5{
   i = (shareN*shareSize);

   sw1 = w1S;
   sw2 = w2S;
   swres = wresS;

   xi = sw1*nSharesPP5;
   xi = xi*shareSize;
   xi = xi + i;

   yi = sw2*nSharesPP5;
   yi = yi*shareSize;
   yi = yi + i;

   resi = swres*nSharesPP5;
   resi = resi*shareSize;
   resi = resi + i;

   aux = xi*8;
   aux2 = st;
   aux3 = aux2 + aux;
   val1 = aux3;

   aux = yi*8;
   aux2 = st;
   aux3 = aux2 + aux;
   val2 = aux3;

   aux = resi*8;
   aux2 = st;
   aux3 = aux2 + aux;
   resval = aux3;
```

```
    add_wrapper(val1,val2,resval);
  }
```

Code Block 4.2: A snippet of code of the add call that uses wrappers.

Aside from the zK field for zero knowledge, an implementation using a small finite field of 101 for testing purposes and an implementation using a finite field of 2 for Boolean circuits were also developed. In each of these cases, the only code that needed to be changed was the wrapper code and global parameters (namely the shareSize), making it simple to swap finite fields for different purposes.

So the Maurer implementation that uses a finite field of 2 is compatible with Boolean circuits, the addition and multiplication calls were changed to perform the XOR and AND operations respectively.This only necessitated a change in the wrapper's code. This specific implementation will further be referred to as the Maurer Boolean circuit implementation.

```
inline
fn add_wrapper(reg u64 x,reg u64 y) -> reg u64{

 reg u64 z;
 reg u64 out;

 x ^= y;
 out = x;

 return out;
}
```

Code Block 4.3: The add wrapper acting as a XOR operation in the Boolean circuit implementation

4.2.5 Memory

As previously stated, the jasmin calls work over a memory space that each party controls. This memory space is divided into two sections: the party's circuit status and the trace.

The circuit status of the party is a memory space consisting of lines for each gate in the circuit containing the party's respective shared values. Depending on the field used, each shared value can have multiple values to describe it.

Meanwhile, the trace memory space contains the remainder of the information generated while computing the circuit. There is reserved space for each gate to store the randomness generated for the operation, which is followed by the messages received by this party for the respective gate. Each one of these messages contain shares received by each party, which in turn

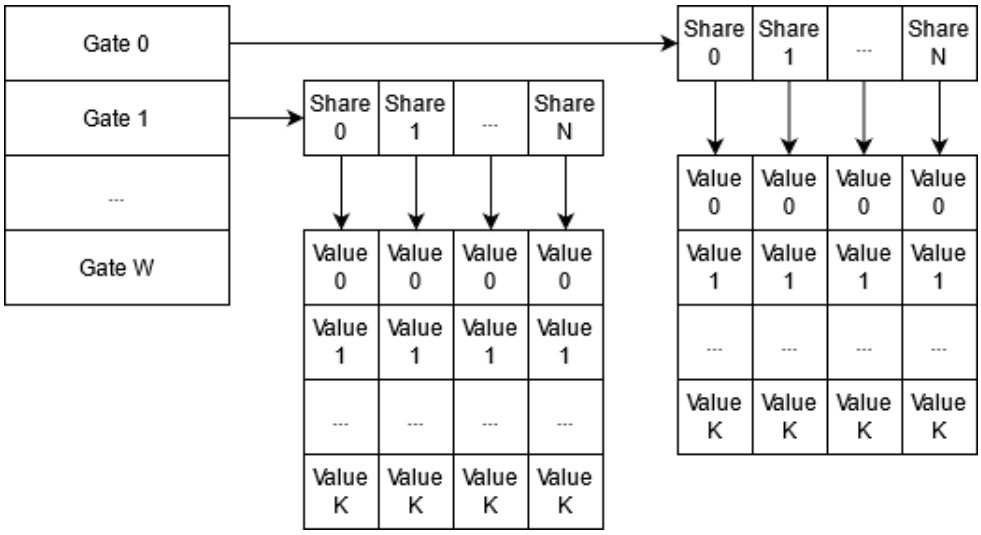


Figure 4.5: Circuit status memory of a party with a circuit of W gates, N shares per party and a field described by K values.

are composed of values that describe a number in a field. Depending on the gate, these may remain empty for the whole circuit. While these values are only used during the computation of each gate and the protocol does not need them for the following gates this trace, however, must be kept because the values required during the protocol may be important for systems that use this implementation as a component (eg: MPC-in-the-head.).

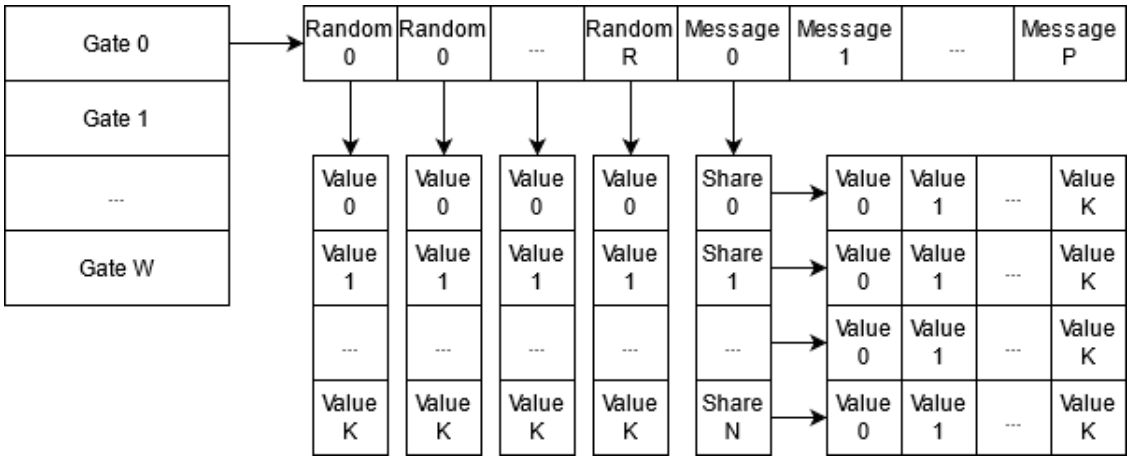


Figure 4.6: Trace memory space of a party with a circuit of W gates, R random numbers per sharing, P parties, N shares per party and a field described by K values.

4.2.6 Scheduler Program Logic

To properly run the protocol and use the jasmin code, it needs to be called from a main program, also known as the scheduler, that each party runs. This program is primarily responsible for memory space allocation, random number generation, and communication with other parties. As previously stated, the jasmin code handles the calculation of shares, message creation and processing, and ensuring values end up in the correct memory space. Using this logic, the main program can use the jasmin calls almost like a "black box", ensuring only that the correct memory pointers are passed to them.

To begin, this program allocates the necessary memory space based on the circuit and Finite Field used, as described in section 4.2.5. After that, the random number generation can be pre-computed during this time and placed in the correct location on the memory space's trace section, or be generated while running the circuit. In any case, the party can begin executing the circuit after this initialization phase.

To execute each gate, the generic approach is to use the _start call (eg: input_start, mult_start), providing it with a pointer to the necessary randomness and another for the messages that are generated. After this, these messages are dispatched between all parties who, depending on the gate, may also be generating and sending messages. After the communications are done and the party has sent and received the required messages, it calls the _end call of the operation, providing it with the pointers to the messages received and the circuit status. The input and multiplication gates are called in this manner, as all of these steps are required to correctly execute the protocol.

In the end calls of each operation, the program knows where to get each gate's message by receiving the current gate index, and then using this value to calculate the respective offset in the trace memory space. Similarly, in operations such as addition and multiplication, this is how the address space containing the values for each index serving as input for the call, and where to write the respective gate output is reached.

```
param int nSharesPP5 = 6;
param int shareSize = 4;

inline
fn getWireIndex5(reg u64 wi) -> reg u64{
 reg u64 res;
 res = wi*nSharesPP5;
 res = res*shareSize;
 res = res*8;
 return res;
}

export //Places shares from message in memory
fn input_end5(reg u64 all_messages status curwire) -> reg u64
{
```

```
  reg u64 wire;
  reg u64 wi;
  reg u64 index;
  reg u64 st;
  inline int i;

  wi = curwire;

  index = getWireIndex5(wi);

  st = status;

  wire = st + index;

  copy_message(wire,all_messages);

  wi = wi+1;
  return wi;

 }
```

Code Block 4.4: An example of the calculation of memory offsets in the input end call.

While this is the main logic behind the jasmin calls, each operation may have slight variations depending on its requirements. The addition and simple multiplication operations do not require communication between parties and can thus be performed simply by passing the current circuit status and which gates are being used as inputs to the add and Smult jasmin calls.

Another slight variation of this execution is found in the public input gate, where while there is no need for communication between parties, as all of them know the value, it is still required for shares to be allocated to each party. The shares for this value and their distribution are created and put in messages in the const_start call, after which the party will choose the one that is associated with their index, and pass it to the const_end call along with the circuit status pointer. As mentioned before, the need for this calling logic is further explained in section 4.3.2. Because the value is known to all parties, these shares are created without randomness, and thus, the const_start call doesn't need it.

Finally, to execute the unsharing, the main logic can be applied to the out_start and out_end jasmin calls, with the only change being that memory for the final output value must be allocated and sent to the out_end call.

4.3 Implementation challenges

A 3 and 5 party scenario implementation, similar to the python prototype, was created. However, unlike the Python prototype, the zk Finite Field required in MPC-in-the-head was also used, allowing this implementation to be used as a component in this scenario. This section describes

the major challenges encountered while developing this implementation, as well as what was done for optimization's sake.

4.3.1 Jasmin programming challenges

While the jasmin programming language allows for low-level programming to further optimize code, it also introduces inherent challenges. Jasmin variables belong to one of three types, address registers, memory stack, or inline. Because the most efficient jasmin variables type is linked to address registers, there is a limit to the number of these variables that can contain values that will be used later in the code. If it is necessary to keep more values than this limit allows, the stack can be used to do so.

Because operations in the Finite Field may require code that uses all of the available address registers, it was required that all values be put into the stack before calling a wrapper. Another risk of indiscriminately using memory registers is that certain operations, such as modulo, require a free register and can cause problems if all of them are already in use. All of this meant that the use of registers in code had to be methodical and well thought out.

4.3.2 Publicly known Variables

Because all parties are aware of publicly known variables or constants, it is unnecessary for them to be secretly shared. However, in order for these to be compatible with all gates without adding extra code and calls, shares must still be created. As previously stated, these shares can be created without the need for randomness if all N shares except the last Nth one are zero and then distributed between parties. While this allows for the use of the add, multiplication, and output calls, some parties end up with shares only containing zeroes. This means that, unless there is another way to save the public value, these parties will no longer have easy access to the original input. This complicates the simple multiplication call, which requires multiplying each share by the public value.

While saving and passing the public values to Jasmin could be a way to solve this issue, it is not the most elegant solution, so instead the creation of the shares can be changed so that each party ends up with at least one instance of the input while still restoring the input when all the shares are added together. This can be accomplished by replacing the first share of each party in a given sharing scheme with the public value, and then filling the remaining shares with the negative of the input in such a way that the unsharing works. For example, in the previously proposed sharing scheme for five parties, with the public value x, the following can be done:

In this example, the shares with index 5, 1, 6, 3 and 7 are always the public input x, the shares with index 0, 2, 4 and 8 are the value -x, and the share with index 9 is the value zero. If these values are unshared, they are added together as $(-x+x-x+x-x+x+x+x-x+0) = x$ restoring the original value x and fulfilling the necessary properties.

	Share 0	Share 1	Share 2	Share 3	Share 4	Share 5
Party 1	x	0	-x	x	-x	x
Party 2	x	-x	0	x	-x	x
Party 3	x	-x	-x	x	0	x
Party 4	x	-x	-x	x	-x	x
Party 5	x	-x	-x	x	-x	x

Table 4.6: Public input x sharing scheme for 5 parties

Making the first share associated with each party be the public value also allows for the code between parties the in simple multiplication call to be uniform between parties. To also maintain this uniformity when computing a constant start gate, all parties' messages are calculated. While this may appear to be redundant, it is required so that the code between all parties remains the same. This eliminates the need for conditional operations, ensures that the design logic is consistent across all functions, and improves performance because conditional operations are more costly than simply populating memory with a value.

4.3.3 Optimization

In order to optimize and make the jasmin calls efficient certain memory-handling strategies had to be implemented in different phases of implementation and in different parts of the code.

One of the most important ideas that had to be implemented in the code was to reduce the number of calls to the stack. Calls to the stack are slower and necessitate instructions to store and read values to a register before using them in various operations. However, because wrappers must be called after putting all future use values in the stack and because there is a limit to the number of available registers, the use of address registers cannot be blind. This means that the use of address registers and stack had to be carefully balanced.

Another method of memory access that was avoided was K in K memory reading. In other words, when memory is accessed in intervals, the code runs slower than when memory is read in order. Because each share has multiple values that describe a number in a field, this error was easy to make in cycles where shares had to be accessed. The code below is an example of this:

```
//Unoptimized share creation
inline
fn create_sharing(stack u64 outS randomnessS){

   (...)
   for i = 0 to shareSize {
         //[5,9,8,7,4,6]
         [out + (0*shareSize+i)*8] = [randomness + (5*shareSize+i)*8];
         [out + (1*shareSize+i)*8] = [randomness + (9*shareSize+i)*8];
         [out + (2*shareSize+i)*8] = [randomness + (8*shareSize+i)*8];
         [out + (3*shareSize+i)*8] = [randomness + (7*shareSize+i)*8];
```

```
        [out + (4*shareSize+i)*8] = [randomness + (4*shareSize+i)*8];
        [out + (5*shareSize+i)*8] = [randomness + (6*shareSize+i)*8];
    }

}

//Optimized share creation
inline
fn copy_share(reg u64 out, reg u64 in){
    inline int i;

    for i = 0 to shareSize{
        [out + i*8] = [in + i*8];
    }
}

inline
fn create_sharing(stack u64 outS randomnessS){

    (...)
    //[5,9,8,7,4,6]
    aux = randomness + (5*shareSize)*8;
    copy_share(out,aux2);

    out = out + shareSize*8;
    aux = randomness + (9*shareSize)*8;
    copy_share(out,aux);

    out = out + shareSize*16;
    aux = randomness + (8*shareSize)*8;
    copy_share(out,aux2);

    out = out + shareSize*8;
    aux = randomness + (7*shareSize)*8;
    copy_share(out,aux2);

    out = out + shareSize*8;
    aux = randomness + (4*shareSize)*8;
    copy_share(out,aux2);

    out = out + shareSize*8;
    aux = randomness + (6*shareSize)*8;
    copy_share(out,aux2);
}
```

Code Block 4.5: Memory access optimization in cycles

Multiple memory access inline functions were created to ensure that optimizations were always performed and to simplify code interpretation. The idea is to always use these when memory values need to be read, copied, stored, or altered in other ways. These auxiliary functions are the following:

- load_array: Receives a share. Writes and returns the values of the share into an array of memory registers.
- load_array_st: Receives a share. Writes and returns the values of the share into an array of memory stack.
- store_array: Receives a memory register containing an address and an array of memory registers. Stores the values in the memory registers in the memory address and subsequent addresses.
- copy_share: Receives two memory register containing addresses. Copies the amount of values in a share starting in the second memory address into the first memory address and subsequent addresses.
- copy_message: Receives two memory register containing addresses. Copies the amount of values in a message starting in the second memory address into the first memory address and subsequent addresses.
- set0: Receives a memory register containing an address. Sets the values of the memory address and subsequent addresses of a share to zero.

4.4 The party-wide scheduler

To properly run and ensure that the implementation was working as intended, a scheduler program focused on calling the jasmin calls was created. This program serves as the scheduler for all parties, and instead of just managing one party's memory and messages, it manages and allocates memory for all parties.

As previously stated, the communication aspect of the protocol is not the primary focus of this implementation, so this party-wide scheduler ignores this part of the MPC and focuses on verifying the validity of the shares messages and memory status. A separate individual party scheduler with simple communications was also created to ensure that each party could execute the protocol without the scheduler having any control over the memory of the other parties.

Much like running a normal individual scheduler, this program executes a circuit gate by gate. One of the main differences, as mentioned before, is that this program controls all of the parties memory, including both the circuit statuses and the trace.

The scheduler allocates two large memory spaces, one after the other, containing all of the parties' circuit status, and another for the traces. Because the traces save the information used in the protocol to create the shares and messages, they can later be used to verify the values in the messages and the circuit status of each party.

Following that, the program follows standard scheduler logic, with the main difference being that it must run each gate for all parties, and thus pass the correct memory addresses containing

each party's status and trace to the jasmin calls. This entails either calculating the correct offsets and adding them to the base memory spaces or separately saving each party's memory space address. The first option was chosen in the program to make the code cleaner and more readable.

Because everything was done locally, messaging in this program worked by simulating communication between parties. As if they had sent each other's messages, each party's message is distributed and placed in the other's trace memory space. The trace memory space is then sent, containing the party's messages received, to the jasmin end calls as usual.

To make the Schedulers processes run more smoothly some extra calls and functions were created. These were created primarily to simplify the manipulation of memory and messages by all parties so that tests and analysis or verification of the jasmin's outputs could be performed easily and so a working proof of work of how to use the jasmin functions could exist.

4.4.1 Memory management and offset Macros

The Scheduler's main functions are managing the parties' circuits' status and Trace memory spaces, as well as sending the correct memory addresses to the Jasmin code. Several memory management functions and memory address offset macros were created to assist with this. For these functions to work several global values are also kept, which are described as follows:

- *wireSize*: The size of the circuit;
- *nSharesPP*: The number of shares each party has per circuit wire;
- *nShares*: Total number of shares per secret value;
- *nParty*: Total number of parties;
- *shareSize*: The number of values that each share needs to be represented;
- *curwire*: The current wire being evaluated, starts at 0. Not important for the offset macros but important for the execution of the circuit.

The scheduler also keeps two main memory addresses, which contain the base addresses of all parties' circuit addresses and traces. When the scheduler needs a specific memory address containing a message or wire status to send to a jasmin function, it uses one of the macros to obtain a value to offset these base memory addresses to the required one.

Using the global values described earlier, the memory space needed to allocated for the memory addresses is obtained by calculating $nParty * wireSize * nSharesPP * shareSize$ times the size of each value in a share for the circuit status.

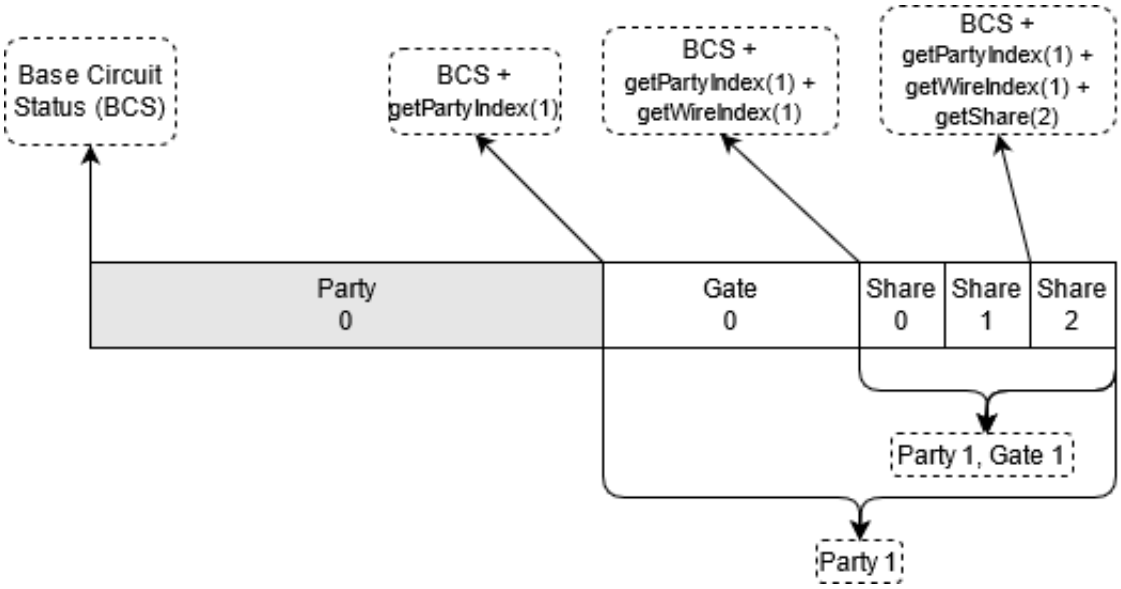


Figure 4.7: Using the memory offset macros to reach a specific share.

For the Trace, $nParty*(wireSize+1)*(((nShares-1)*shareSize)+(nParty*nSharesPP*shareSize))$ times the size of each value in a share needs to be allocated. The trace memory space needs to be multiplied by $(wiresize+1)$ instead of just $wiresize$ to allow space for the messages created and received in the final output gate. The final multiplication factor is divided in two parts the first, $((nShares-1)*shareSize)$, containing the randomness generated, and the second, $(nParty*nSharesPP*shareSize)$, the messages received in each wire.

Because this is an important part of the Scheduler, some of the code will be described to better explain how this memory is accessed. There are three of the before mentioned offset macros for the circuit status memory:

- *getPartyIndex*: Receives a party's index and returns the offset needed to add to the base circuit status pointer to reach this specific party's circuit status memory address;
- *getWireIndex*: Receives a wire index and returns the offset needed to add to a specific party's circuit status memory to reach a status wire's memory address;
- *getShareIndex*: Receives a share's index and returns the offset needed to add to a specific status wire's memory address to reach that share's memory;

```
int getPartyIndex(int p){
  return (p*(shareSize*nSharesPP*wireSize));
}

int getWireIndex(int w){
  return w*(nSharesPP*shareSize);
}
```

```
int getShareIndex(int s){
  return s*(shareSize);
}
```

Code Block 4.6: Memory offset Macros for the Circuit Status Memory

Similarly, offsets Macros for the memory containing the Trace of all parties were created, and they are as follows:

- *getTracePartyIndex*: Receives a party's index and returns the offset needed to add to the base trace pointer to reach this specific party's trace memory address;
- *getTraceWireIndex*: Receives a wire index and returns the offset needed to add to a specific party's trace memory to reach a trace wire's memory address;
- *getTraceMessagesIndex*: Gets the offset needed to add to a trace wire's memory adress to reach the pointer containing the messages received in this wire. This is needed because this address points to the randomness used in this wire;
- *getMPartyIndex*: Receives a party's index and returns the offset needed to add to the pointer messages to reach the message received from a specific party;
- *getShareIndex*: Receives a share's index and returns the offset needed to add to a message to reach that share's memory. The same as in the circuit status memory.

As is shown in Code Block 4.6 these macros are simple calculations of offsets whose values depend on the circuit and party's involved. A full example of the use of these macros to go trough the whole circuit status of all parties in order to print it can be found in Code Block 4.7 on section 6.1 - Analysing outputs and results. Code block 4.8 contains an example of the use of offset macros for the Trace. Figure 4.7 shows a representation of how to reach a specific share located in the memory using these.

```
void print_statuses(uint64_t* statuses){
  for(int p = 0;p<nParty;p++){
    printf("\nStatus %d:\n",p);

    uint64_t* status = statuses + getPartyIndex(p);

    printf("CurrentWire: %ld\n",curwire);

    for(int w = 0;w<wireSize;w++){

        uint64_t* wire = status + getWireIndex(w);

        for(int s = 0;s<nSharesPP;s++){

          uint64_t* share = wire + getShareIndex(s);
```

```
          printf("(");
          for(int sp=0;sp<shareSize;sp++)
            printf("%ld,",share[sp]);

          printf(") ");

        }
        printf("\n");
    }
  }
}
```

Code Block 4.7: Using macros to print the circuit status of all parties.

While there are other methods for managing and locating the correct memory addresses for the jasmin functions, this is the one that is used because it follows the logic described in the previous chapter and how the jasmin functions were developed with minimal effort and overhead.

To make jasmin calls, the correct memory addresses with the correct values must be passed to the calls.This is especially noticeable in calls that require randomness and a memory location to write outgoing messages to other parties. The memory addresses used to store these should be those in the Trace memory space, which can be calculated using the memory offsets. The code in Code Block 4.8 describes how to call input_start.

```
void do_input(int partyIndex, uint64_t* input,uint64_t* statuses, uint64_t*
    tracePP){

  int wireIndex = getNumWiresDone();

  uint64_t* traceParty = tracePP + getTracePartyIndex(partyIndex);

  uint64_t* traceWire = traceParty + getTraceWireIndex(wireIndex);

  uint64_t* traceMessages = traceWire + getTraceMessagesIndex();

  uint64_t* r = traceWire;
  init_randomness(r);

  uint64_t* jasminMessages = traceMessages;


  input_start5(input,jasminMessages,r);

  do_dispatch(tracePP);

  do_input_end(statuses,tracePP,partyIndex);


  curwire++;
```

```
}
```

Code Block 4.8: Calling input_start from the Scheduler.

The scheduler also obtains the correct memory addresses from the offsets macros to write the randomness and outgoing messages from jasmin to, as seen the same Code Block 4.8. It then simulates message transmission by using the Dispatch call described in section 4.4.2. Finally, after all of the messages have been correctly placed in the Trace, the input end call is executed to populate the circuit status memory. The execution of most gates in a local party scheduler follows this logic with minor variations depending on the operation (e.g., the multiplication gate must run the mult_start call for all parties, the add call does not require messages or randomness, and so on).

4.4.2 The Dispatch Function

As was described before the scheduler runs all parties locally. Despite this, Jasmin continues to generate messages that are intended to be sent and received by multiple parties. One approach would be for the scheduler to handle each message and place it in the appropriate memory address for each party.

While this was the method used in the first versions of the Scheduler, it was eventually decided that a new jasmin function would be created not only because this would be a common problem when verifying and running the jasmin code in locally executed scenarios, but also due to these scenarios being commonplace (E.g: when integrating and testing the usage of jasmin functions in a new setting or system).

In a locally executed scenario, this jasmin function, called dispatch, is responsible for distributing and placing messages created in other jasmin functions in the correct memory space of each party by simulating communication between parties.

The dispatch jasmin call receives memory address pointers for each of the parties, which contain the messages created in other _start calls that are intended to be distributed between participants. The call's algorithm works as follows:

1. Check the messages in the party's respective address beginning with the party with the lowest index.
2. If the message isn't meant to be received by this party, switch it with the message with the current party's index owned by the party that should receive it.
3. Repeat for each party until all messages have been properly placed. As the algorithm progresses through the parties, the number of messages that must be switched per party

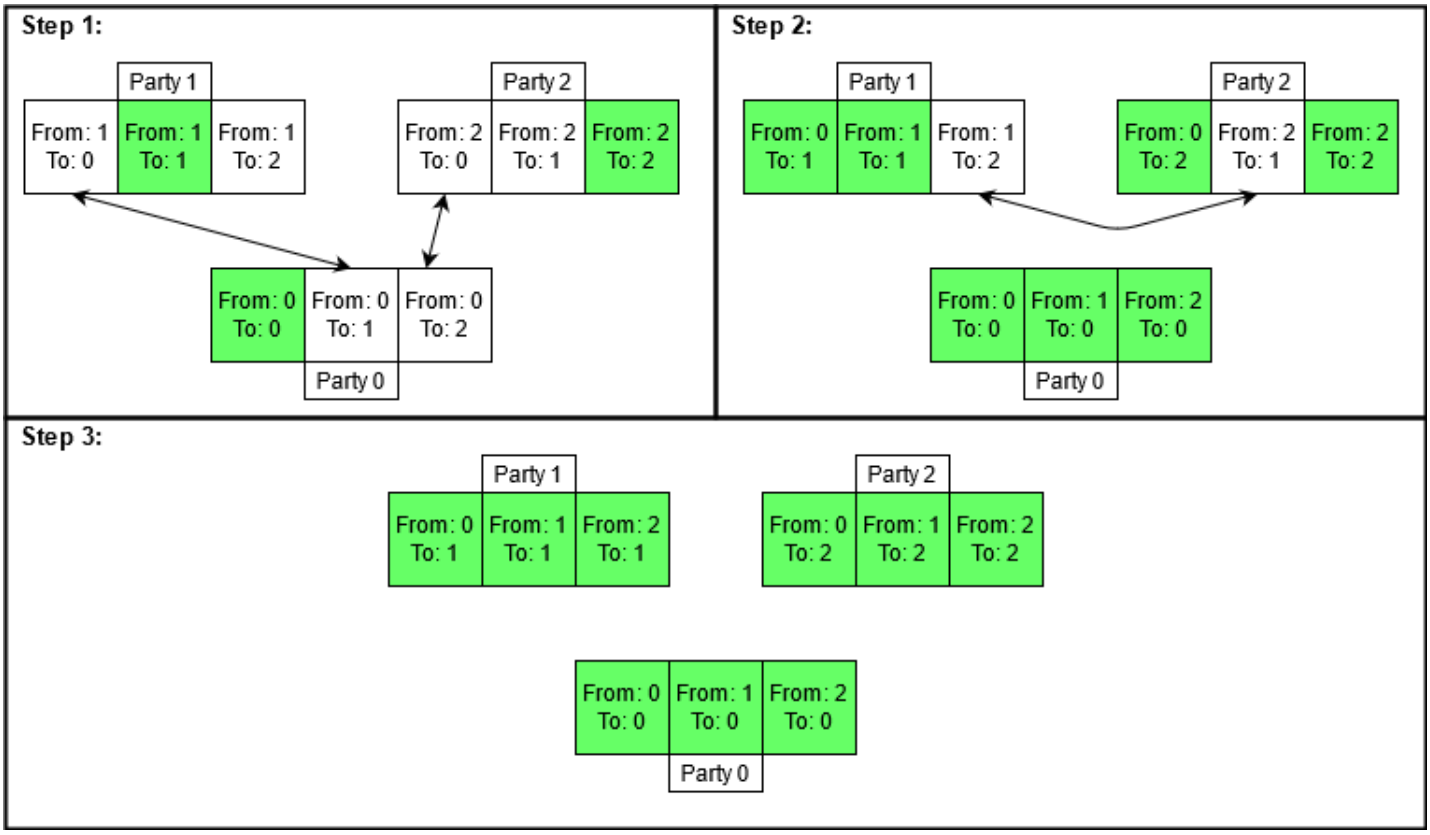


Figure 4.8: Running the dispatch call with 3 parties.

decreases.

Figure 4.8 shows an example of this simple algorithm running with 3 parties. While conditionals and loops can be used to make this call compatible for any number of parties, this approach was avoided in order to make the implementation as efficient as possible. As a result, a dispatch hard-coded for the 5 and 3 party scenarios was created, ensuring as little overhead as possible.

Similarly, using dispatch from the scheduler would need to be hard-coded in order to keep overhead to a minimum. Aside from that, the only thing the party-wide Scheduler needs to do now to correctly place the messages in the Trace is to use the macro offsets to calculate and send the messages to the dispatch jasmin call, as is shown in Code Block 4.9.

```
void do_dispatch(uint64_t* tracePP){

  int wireIndex = getNumWiresDone();


  uint64_t* traceParty = tracePP + getTracePartyIndex(0);
  uint64_t* traceWire = traceParty + getTraceWireIndex(wireIndex);
  uint64_t* messages0 = traceWire + getTraceMessagesIndex();
```

```
  traceParty = tracePP + getTracePartyIndex(1);
  traceWire = traceParty + getTraceWireIndex(wireIndex);
  uint64_t* messages1 = traceWire + getTraceMessagesIndex();


  traceParty = tracePP + getTracePartyIndex(2);
  traceWire = traceParty + getTraceWireIndex(wireIndex);
  uint64_t* messages2 = traceWire + getTraceMessagesIndex();


  dispatch(messages0,messages1,messages2);

}
```

Code Block 4.9: Calling dispatch from the Scheduler

Chapter 5

GMW Development

Until now, all of the design, code, and challenges described in this work have been centered on the Maurer protocol's implementation. However, another secure MPC protocol was also implemented to investigate which designs philosophies could be applied or adapted across multiple protocols and to have an alternative to run MPC in a Boolean circuit scenario. The GMW described in the Background chapter was chosen for this purpose.

This chapter will discuss not only present the GMW implementation's design but also the differences between the implementation of the Maurer and GMW protocols, the specific challenges encountered with this protocol, and the purpose of the jasmin calls and outputs developed.

5.1 Design

Because the Maurer implementation provided some MPC implementation experience, and because the GMW protocol has a simpler sharing scheme that does not require the pre-computation of values such as multiplication pairs, a preliminary prototype in a different language was deemed unnecessary.

The full program, as with the previous implementation, is divided into jasmin calls responsible for message creation and memory manipulation, and a master program called Scheduler that will allocate memory and use these jasmin calls to run the circuit.

Besides this, an important detail inherent to this protocol stems from the fact the GMW protocol is focused on Boolean circuits, and as such, large finite fields to represent shares are not needed. As a result a share will always be able to be represented by a single value, as shares will always be values 0 or 1.

5.1.1 Circuit representation

The circuit representation, which can be found in the previous chapter, was the main design decision carried over, allowing most of the same logic to be applied when dealing with gates and shares. As a result, some design decisions resulting from this representation will follow similar patterns, and while comprehension of the previous chapter is not required to understand this one, understanding the previous implementation will aid in fully grasping all concepts. The following gates will be the focus of this implementation:

- Input gates, for each party's secrets;
- XOR gates, to perform a XOR operation between two other gate values;
- AND gates, to perform a AND operation between two other gate values.

As with the previous chapter these gates can be represented in text with parameters such as *(input,2)* for an secret input gate provided by the party with index 2, *(XOR,2,3)* for a xor operation between values obtained from gate with index 2 and 3, or *(AND,2,3)* with the same logic as the XOR gate.

5.1.2 Messaging

In order to compute the Input and AND gates the parties must communicate certain values as per the protocol. As such the following messaging system was implemented.

For the input gates, the party sharing the secret value must create a message with a share for each other party, and then obtain their share from the messages created, according to the protocol. As a result, a party will send a single message to all parties for each input gate.

In the AND gates more complex messages must be created and sent to the Oblivious transfer protocol. Following the protocol, each party must create a message for each other party that contains two two masked values (r and $m = a_i \oplus r$). When receiving a message containing these two values, each party will obtain one of them the as per the Oblivious Transfer protocol and use it to calculate their share for the gate with the other received messages.

These more complex messages have some challenges and ramifications inherent to them that will be explored in sections 5.1.4 Memory and 5.2 Implementation challenges.

5.1.3 Jasmin calls

As with the previous implementation, the following jasmin external calls were created to handle message creation and circuit status modification:

- input_start: Called when starting an input gate. Creates the input messages to be distributed between players. Receives a secret input, the necessary random numbers to create shares and a space in memory to write the messages to send to all players;
- input_end: Called after having received the respective input's message from input start.Populates the party's status with the input's shares. Receives the space in memory containing the party's messages, the one containing the circuit status and the index of the current gate;
- xor: Called when computing a Xor gate. Performs the XOR operation on the two wires' values and populates the party's status. Receives the space in memory containing the circuit status, the index of each of the gates being added, and the index of the current gate;
- and_start: Called when computing an And gate. Creates the messages required for the 1-out-of-2 Oblivious Transfer as per the GMW protocol and calculates $a_i \wedge b_i$. Receives the space in memory containing the circuit status, the index of each of the gates being multiplied, the index of the current gate, the necessary random numbers to create the messages and a space in memory to write the messages to send to all players;
- and_end: Called after receiving all party's And messages. Adds together the messages received to get the party's AND gate's shared result and populates the status. Needs Oblivious transfer to be done. Receives the space in memory containing the party's messages, the one containing the circuit status and the index of the current gate.

Because each party only has one share, the unsharing or output gate is simply accomplished by each party making their final circuit share publicly known and then reconstructing the final output by XOR'ing all the values together. Unlike in Maurer, where all of the unsharing must be organized in order to obtain an instance of each share, unsharing only requires a simple operation, eliminating the need for an output call.

5.1.3.1 Recurring parameters

Some recurring parameters and values are present in jasmin code blocks, like in the Maurer's implementation code these are constant values that depend on the protocol or party size, and are relevant for message creation or memory manipulation.

The most common is nShares or NParty, which is the total number of shares, which is the same as the total number of parties in the GMW protocol because each party always keeps one independent share for any given value.

The value 8 may appear in code again because it is the size that an unsigned integer takes up in memory. However, unlike the Maurer protocol, there is no need to draw special attention to the number of shares each party owns per gate or the number of values each share contains in the code, as they are always both 1.

5.1.4 Memory

The memory space was chosen, as in the previous implementation, to be divided into a party circuit status, which contains the shares that each party has of each gate, and a trace, which contains messages received and random numbers generated for each gate. The structure of these memory spaces, however, differs due to differences in message size and the absence of variable finite fields.

Because the sharing scheme used limits each party to a single share per gate, and because these shares can always be represented by a single value, the memory's circuit status can simply be represented by an array of shares.

As for the Trace memory space, while the messages created in the input_start call only contain one value, the AND gate messages contain two masked values (r and $m = a_i \oplus r$). This means that, in addition to the space required to store the random numbers generated in each gate, the Trace's messages must also allow for the storage of two values. The Trace is represented in figure 5.1.

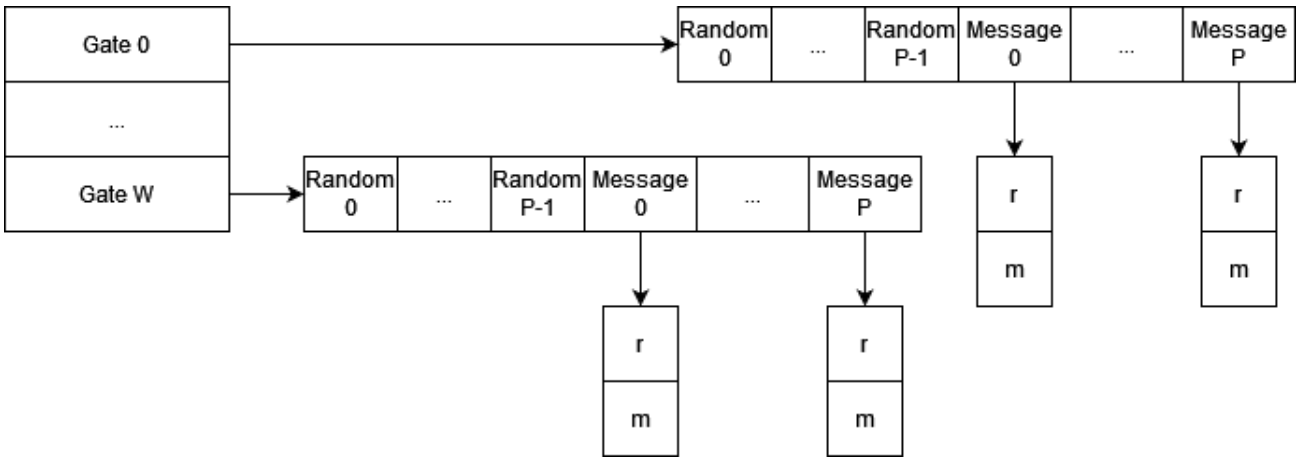


Figure 5.1: Trace memory space of a party with a circuit of W gates and P parties in GMW.

5.2 Implementation challenges

When implementing this protocol, some of the issues encountered in Maurer's implementation had to be addressed again, but they had already been discussed in the previous chapter. Among these the jasmin programming challenges and optimization strategies were mostly the same. While most call's implementation was straight forward, the AND calls and message creation provided some more specific challenges.

In order to create the messages necessary for the AND operation, each party needs to calculate

three types of values: the XOR of all the randoms used, the pair $a_i \wedge b_i$ from a AND wrapper, and the masked messages for 1-out-of-2-OT for each of the other parties. Furthermore, there needs to be a memory space to keep all these values. The rest of this section will describe the process done for an AND gate of $a \wedge b$ shared values.

As previously stated, we can solve the memory space issues by increasing the Trace message size for this protocol from one to two values per party. The random bit r can be kept in the space for the first share, while the masked value m is kept in the space for the second. The other two values necessary for the protocol (the XOR of all random values used and $a_i \wedge b_i$) can be XOR'ed together here, as they will be in the end, to save one memory space. In any case, there are two memory spaces free in the message trace because, unlike the Maurer Protocol, a Party does not need to create a sharing for itself, or in other words, send a message to itself. This space can be used to simulate a party sending the other two values' as a message to itself to be used in the and end call instead. Figure 5.2 describes the output arrays for the first two parties in a five party scenario and the two free memory spaces being used for these two extra values.

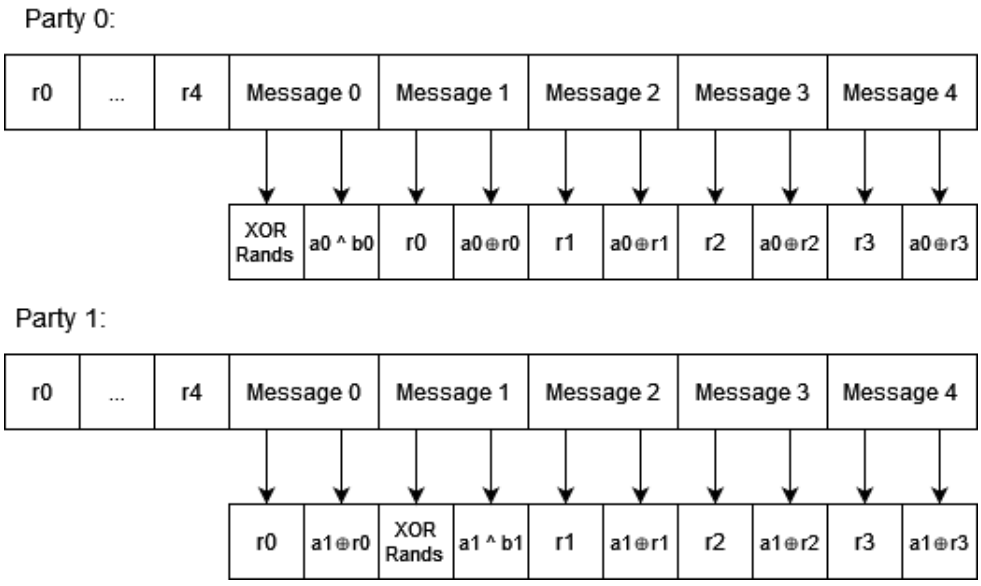


Figure 5.2: Output for the first two parties of the and_start call in a five party scenario

As previously stated, the two values could be XOR'ed together at this phase and kept in one of the two addresses, but they are kept separate so that all of the calculated values are available and the difference in occupying this one memory space is negligible. Following that, these messages will be sent to all other parties using an Oblivious transfer protocol and will be selected based on the Party's share of the b value. Similarly, messages are received via OT and are selected based on the local b's share. All of these values are then passed to the and_end call, which XORs them all together and stores the resulting share in the correct location in the party status memory.

```
//create messages of OT2 for other players (ai&bj)

   rcount = 0;

   for i = 0 to 5 { //5 = nParty
      rd3 = partyIndex;
      if(rd3 != i){

                  aux2 = i*2; //2 = number of messages per party
            aux2 = aux2*8;
            aux1 = out;
            aux1 = aux1 + aux2;
            z = aux1;

            rd1 = randomness;
            aux2 = rcount;
            aux2 = aux2*8;
            aux1 = rd1 + aux2;
            y = aux1;

            aux1 = z;
            aux2 = y;

            [aux1] = [aux2];

            aux1 = aux1 + 8;
            z = aux1;

            XOR_wrapper(y,x,z);

            aux1 = rcount;
            aux1 = aux1 + 1;
            rcount = aux1;
      }
   }
```

Code Block 5.1: Creating messages for 1-out-of-2-OT

5.3 The party-wide Scheduler

As in the case of the Maurer implementation, a party-wide scheduler for the GMW protocol exists for the same reasons. Because most of the main points of such a scheduler were covered in the previous chapter, this section will concentrate on the differences between the two.

The main differences between a party wide scheduler or program that uses GMW jasmin calls are in memory space creation and communication between parties. The majority of these differences can be resolved by using the same logic described in the GMW section of the

development chapter for the jasmin code of adjusting global parameters and values and using them to modify macros and memory allocation.

Much like the jasmin code, the values represented by *nSharesPP*, *nShares* and *shareSize* in the Maurer Scheduler are redundant in this protocol. The following global values are kept and are refereed to in the scheduler code:

- *wireSize*: The size of the circuit;
- *nParty*: Total number of parties;
- *curwire*: The current wire being evaluated, starts at 0. Not important for the offset macros but important for the execution of the circuit.

In a party wide scheduler, the total memory space needed to be allocated for all of the parties' circuit status memory, including output gates, is $(nParty * wireSize) + nParty$. Similarly, the space for the trace memory can be obtained from a much simpler $nParty * (wireSize + 1) * ((nParty - 1) + (nParty * 2))$ due to the smaller and simpler messages.

This implementation employs the same strategy of using offset macros to access specific parts of memory. They are as follows:

- *getPartyIndex*: Receives a party's index and returns the offset needed to add to the base circuit status pointer to reach this specific party's circuit status memory address;
- *getWireIndex*: Receives a wire index and returns the offset needed to add to a specific party's circuit status memory to reach a status wire's memory address;
- *getTracePartyIndex*: Receives a party's index and returns the offset needed to add to the base trace pointer to reach this specific party's trace memory address;
- *getTraceWireIndex*: Receives a wire index and returns the offset needed to add to a specific party's trace memory to reach a trace wire's memory address;
- *getTraceMessagesIndex*: Gets the offset needed to add to a trace wire's memory adress to reach the pointer containing the messages received in this wire. This is needed because this address points to the randomness used in this wire;
- *getMPartyIndex*: Receives a party's index and returns the offset needed to add to the pointer messages to reach the message received from a specific party;

It is also worth noting that when debugging the implementation of a new scheduler or integration, the memory space may be filled with zeroes by default, which is relevant and may confuse a debugger in a boolean setting. As a result, the memory in the created party wide scheduler was filled with an impossible value such as 9. Aside from that, the visualisation of the

memory is simplified due to the lower number of shares and smaller messages, as shown in figure 5.3.

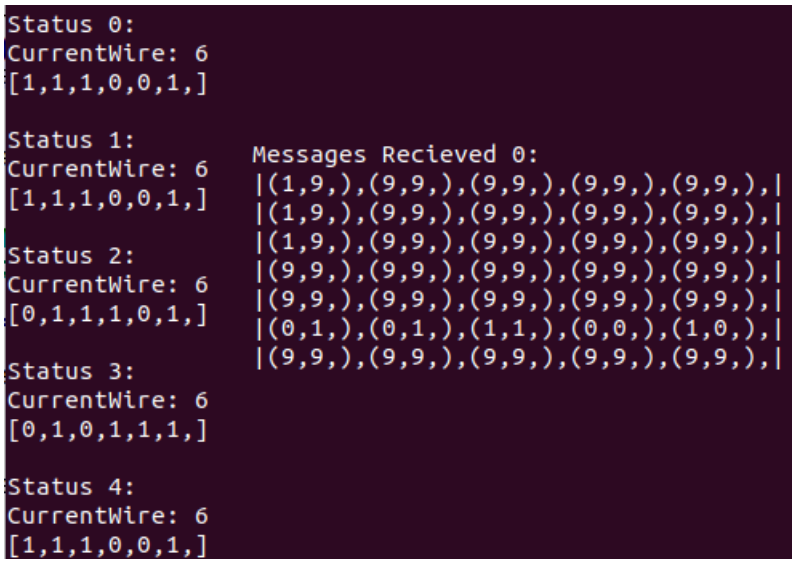


Figure 5.3: Visualising the memory of the GMW scheduler. Pictured are the circuit status of all parties and the messages received of the first party before Oblivious Transfer.

The requirement for an Oblivious Transfer protocol for message exchange is the more significant difference from the Maurer's scheduler. To ensure safety, a third-party implementation or library should be used for this part of the protocol. The OTExtension[29] implementation described in the State of the Art chapter is an example of this.

Finally, the lack of a jasmin output call makes the program calling jasmin responsible for the final output reconstruction. As stated in the previous chapter, this can be easily resolved by having the parties exchange their final value of the party circuit status and XOR'ing all of these shares together.

As it can be seen from code block 5.2, the differences in logic and code aren't much different in the scheduler from the Maurer code. The main difference in this function of the party-wide scheduler is the call of the Oblivious transfer after all the parties have created their messages.

```
void do_and(int wire1,int wire2,uint64_t* statuses, uint64_t* tracePP){

    for(int p = 0;p<nParty;p++){

        int wireIndex = getNumWiresDone();

        uint64_t* traceParty = tracePP + getTracePartyIndex(p);

        uint64_t* traceWire = traceParty + getTraceWireIndex(wireIndex);
```

```
        uint64_t* jasminMessages = traceWire + getTraceMessagesIndex();


        uint64_t* r = traceWire;
        init_rand(r);

        uint64_t* status = statuses + getPartyIndex(p);

        and_start5(status,(uint64_t) wire1,(uint64_t) wire2,jasminMessages,r,p);

    }

   do_OT(tracePP);

   do_and_end(tracePP,statuses);

   curwire++;
}
```

Code Block 5.2: Calling and_start from the party wide Sheduler for 5 parties.

Chapter 6

Experimental Results

This chapter will go over all of the analysis and results obtained from testing, benchmarking, and verifying the MPC implementations that were created. If omitted, the described implementation will be an implementation of Maurer's MPC using the Zk finite field and a party-wide scheduler.

The strategies and analysis methods will be described as well, with the goal of making it easier to integrate the jasmin code into new systems and programs.

6.1 Analysing outputs and results

While testing and verifying the use of the jasmin operations, erratic behavior or incorrect results may necessitate checking the values of the circuit status in memory and the messages passed between parties. If the scheduler is used correctly, all of these values will remain in memory throughout the circuit's execution and can be verified.

While printing and examining these values are simple in execution, they are critical for integrating jasmin operations in a new setting, and as such, the print and methods for properly reading these values will be described in this section.

The two main functions found in the scheduler code used for displaying all variables currently in memory are *print_statuses*, (described in Code Block 4.7) and *print_tracePP* for the circuit status and trace respectively. These functions use the macros described in section 4.4.1 to go through memory value by value and display them in a structured manner. These functions are also significant because they can be used to lay the groundwork for correctly traversing memory for verification purposes.

The information displayed for the circuit status is divided into parties, each with wires separated by line breaks. The shares of each wire are also separated by parenthesis, with each group containing the values that represent the share. Knowing which gates have been evaluated may be useful information, so the *currentWire* value is also displayed. A visual representation is show in figure 6.1.

```
Status 1:
CurrentWire: 0
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,)
```

Figure 6.1: Visualising the empty circuit status of Party 1.

Similarly, when viewing the information in the trace memory, the parties and wires are separated. The values of the randomly generated numbers for this gate are displayed first in each wire, separated by parenthesis as in the circuit status print. Following the display of the random values, the messages received in this gate are displayed, each separated by a vertical line. The messages are listed in the order of the party index. Each message contains shares that are separated into grouped values in parenthesis once more. A visual representation is show in figure 6.2.

```
TRACE PARTY0:


Wire 0:

Randoms:
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0
,0,0,0,) (0,0,0,0,) (0,0,0,0,)

Messages:
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) |
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) |
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) |
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) |
(0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) (0,0,0,0,) |
```

Figure 6.2: Visualising the empty Trace of the first wire of Party 0.

After ensuring that the output calls are correctly integrated into the scheduler, another method for further verifying each gate's values is to use the output calls as a verification method. While their primary purpose is to implement the final output gate, the calls have been designed in such a way that calling them in the middle of the circuit has no effect on the share values or the final results. As a result, when executing a circuit, the output gate can be used to obtain the result of any gate. However, because the parties in the trace must communicate in order to

produce the output, this has the unintended consequence of generating messages between them, which may cause unexpected behavior if the values in the trace are used elsewhere.

Alternatively, the scheduler itself can have a function that reconstructs the value of a gate similar to the output gate using the parties circuit status memory for testing and verification purposes. However, this necessitates that the implementation be done in the scheduler.

6.2 Testing Circuits

Many tests were used during the development of the jasmin calls to test the various functionalities. This section will explain which circuits and values were chosen and why, so that they can be used when integrating or adapting this implementation to verify its correct usage. The circuits shown in this chapter follow the representation described in the Development chapter.

Testing each operation in very small circuits with a variety of values is encouraged and was done before using the following circuits to better understand the values and outputs from memory and to catch errors more easily with this understanding.

Multiple gates of that operation should be used to test each operation, with the input gate ensuring that each party provides at least one input with different values to catch potential mistakes. Besides this, following initial testing with a circuit using randomly generated input values while verifying the expected circuit result was also a strategy used, is useful for detecting more hidden unexpected behaviors and is also good practice. For example, to test the Smult operation the circuit *{(input,0), (input,1), (input,2), (input,3), (input,4), (addition,3,4), (Smult,0,5), (Smult,0,6), (Smult,0,7), (Smult,0,8)}* was utilised, after properly testing the addition and input operations.

A simple circuit used to test 5 parties is described by the gates *{(input,0), (input,1), (input,2), (input,3), (input,4), (addition,3,4), (multiplication,0,5), (multiplication,0,6), (multiplication,0,7), (multiplication,0,8)}*. The main purpose of this circuit is to serve as an all-purpose tester capable of quickly reaching very large values that require shares represented in a large field to avoid overflow. Although it doesn't test constant values, this circuit uses at least a little bit of each other gate, including an input gate provided by each party, an addition and multiple multiplications.

In the implementation created an example of such a field that was used is the Zk-field. This field was implemented by using 4 unsigned integers to represent a number, and as such, shares where represented by 4 variables (for the Scheduler this meant that the global variable shareSize would be 4).

The first input values used to test the Zk field with this circuit where represented by the groups of unsigned integers *(10000, 0, 0, 0), (20000, 0, 0, 0), (30000, 0, 0, 0), (40000, 0, 0, 0), (50000, 0, 0, 0)* for each one of the input gates respectively. These values ensure that the circuit's result can't be represented by just one unsigned integer. The expected result is the value represented in Zk by *(14556284461941522432, 48, 0, 0)*.

	Word 1	Word 2	Word 3	Word 4
In 1	10327709304191855024	10926837314174586968	11179987232314788621	2368260955130346714
In 2	12072847408584426164	17667690894698371173	16612283213309483868	6916042084488537607
In 3	10204008385840067079	12982271688899101709	15380047305533896926	2530675914559331311
In 4	18012934314358021182	17338994251328739453	13254565268838465264	871387676949938648
In 5	6765488442056923073	15903439452221252288	1345687455747554025	1222244449942681894
Out	8229339962084616515	5926938253005890197	10568747782583765907	4917644870093405243

Table 6.1: An example of testing values for the Zk Field

The next step was to test with randomly generated values in this circuit to further ensure the Zk library used in the jasmin wrappers was being utilized correctly. To accomplish this, a simple program was written that generated random values in the Zk field in SageMath, a mathematics software system[28]. After this, another one was also created that applied the operations in the test circuit to these test values. SageMath is a great language for creating these values because it allows for very easy manipulation of large numbers. Code Block 6.1 contains an example of these type of programs for the Zk field.

```
p = 2^255-19

Fp = GF(p)

ZZx = int(Fp.random_element())
print("Original number: " + str(ZZx))

w0 = int(mod(ZZx,2^64))
w1 = int(mod((ZZx // 2^64),2^64))
w2 = int(mod((ZZx // 2^128),2^64))
w3 = int(ZZx // 2^192)

print("Values that represent number: (" + str(w0) + ","+ str(w1) + ","+ str(w2)
    + "," + str(w3) + ")")

FPx = Fp(w0 + w1*2^64 + w2*2^128 + w3*2^192)
print("Reconstructed number: " + str(FPx))
```

Code Block 6.1: Creating random Zk field values in SageMath

Table 6.1 shows one example of the input values for each party and expected output used for testing the use of this field for the circuit described in this section.

6.3 Message bandwidth and overhead

As part of the integration into new systems, it may be useful to understand the details of the messaging system created by the jasmin implementations, especially since the scheduler or

program calling the code is responsible for implementing the protocol's communication portion. As a result, this section will go over important aspects of the messages generated by jasmin.

The maximum amount of bandwidth, or amount of data, occupied by messages in all jasmin implementations can be calculated by using the trace memory allocation formula in each respective scheduler and removing the space meant to store the random numbers. For example, in an Maurer implementation, from $(wireSize + 1) * (((nShares - 1) * shareSize) + (nParty * nSharesPP * shareSize))$ for the trace of a single party, $(wireSize + 1) * (nParty * nSharesPP * shareSize)$ can be obtained. However, because one of the messages created by jasmin is meant to be used by the executing party and doesn't need to be sent, the actual network bandwidth is $(wireSize + 1) * ((nParty - 1) * nSharesPP * shareSize)$ [1]. This formula gives the number of maximum possible bandwidth used for messages for a circuit of *wireSize*, where all gates would be multiplications, and allowing for an extra output gate.

As described before, in a multiplication gate a party will send messages to all other parties, making the total number of messages sent between every party be $nParty * (nParty - 1)$. In an input gate, only the party that is sharing the secret send messages, making the total number of messages sent in this type of gate be $nParty - 1$.

In an GMW implementation the same logic can be followed, making the maximum amount of message bandwidth for a single party in a given circuit be $(wireSize + 1) * ((nParty - 1) * 2)$ obtained by removing the space for random numbers from $(wireSize + 1) * ((nParty - 1) + (nParty * 2))$. This is once again assuming the worst case scenario where every gate is an addition gate and two messages for the Oblivious transfer need to be generated by each party, for each party.

As such, the total number of messages sent in an addition gate would be $nParty * (nParty - 1)$ because, while the Oblivious transfer requires two values, they can be sent in the same message. This way, each party sends a larger message with the Oblivious Transfer protocol to each other party. The input gate messaging in GMW follows the same logic of the Maurer implementation, and as such, a total of $nParty - 1$ messages are sent.

6.4 Formal Verification

The Maurer implementation's jasmin code underwent formal security verification, which followed the standard process of verification using jasmin's tools. This procedure entailed extracting the code into EasyCrypt and then creating formal security proofs. The publication Formally Verified Security Proofs and Implementations of MPC-in-the-Head[4] describes this procedure in more detail.

[1] nParty -> Number of parties; nShares -> Total number of shares; nSharesPP -> Shares per party; wireSize -> number of gates in the circuit; shareSize -> number of values per share.

6.5 Benchmarks

A number of benchmarks on the execution of jasmin calls and gate computation in schedulers were performed to further test and analyze the code produced. The obtained results are described in this section.

6.5.1 Maurer with 5 parties

The first set of benchmarks were performed on a local scenario with a wide-party scheduler to analyze the main Maurer implementation using the zK finite field, used to be integrated as a zero knowledge proof component. All the times displayed in this section and the following are represented in seconds.

5 parties:	add call	addition gate
Time 100 000 wires	0.0064134	0.031188
Time 10 000 wires	0.000608	0.003161
Time 1 000 wires	0.000061	0.000334
CPU Cycles Median	160	794
CPU Cycles Average	185	878

Table 6.2: Benchmarks for an addition gate by 5 parties.

Table 6.2 displays the results of repeatedly executing an addition gate 100 000, 10,000, and 1000 times. The times displayed for the addition gate represent the total amount of time spent in computation by all five parties. As can be seen, the times increase linearly with the number of gates, and the times of the call multiplied by the number of parties coincide with the total execution time of the gate, showing that the scheduler overhead for this gate is minimal.

5 parties:	input_start call	input_end call	dispatch call	input gate
Time 100 000 wires	0.012247	0.001534	0.02211	0.178303
Time 10 000 wires	0.001389	0.000191	0.002097	0.017217
Time 1 000 wires	0.000116	0.000015	0.000198	0.001859
CPU Cycles Median	312	38	551	4459
CPU Cycles Average	345	40	626	5034

Table 6.3: Benchmarks for an input gate by 5 parties.

Table 6.3 displays the results of repeatedly executing an input gate 100 000, 10,000, and 1000 times. The times displayed for the input gate represent the total amount of time spent in computation by all five parties. The same conclusions can be taken as the input gate where the times increase linearly and the scheduler overhead is minimal, e.g.: $(0.012247 + 0.001534 + 0.02211) * 5 = 0.179455 \approx 0.178303$.

5 parties:	mult_start call	mult_end call	dispatch call	multiplication gate
Time 100 000 wires	0.076821	0.020069	0.02211	0.976736
Time 10 000 wires	0.007079	0.002019	0.002097	0.105822
Time 1 000 wires	0.000699	0.000178	0.000198	0.009675
CPU Cycles Median	1959	501	551	24423
CPU Cycles Average	2167	566	626	27308

Table 6.4: Benchmarks for a multiplication gate by 5 parties.

Table 6.4 displays the results of repeatedly executing a multiplication gate 100 000, 10,000, and 1000 times. The times displayed for the input gate represent the total amount of time spent in computation by all five parties. While once again the execution times increase linearly, the scheduler overhead is quite significant, e.g.: $(0.076821+0.020069+0.02211)*5 = 0.595 < 0.976736$.

5 parties:	const_start call	const_end call	constant gate
Time 100 000 wires	0.004445	0.001506	0.033096
Time 10 000 wires	0.000479	0.000144	0.003189
Time 1 000 wires	0.000043	0.000016	0.000298
CPU Cycles Median	113	33	747
CPU Cycles Average	126	42	819

Table 6.5: Benchmarks for a constant gate by 5 parties.

Table 6.5 displays the results of repeatedly executing a public input gate 100 000, 10,000, and 1000 times. The times displayed for the input gate represent the total amount of time spent in computation by all five parties. Besides noting the low overhead of the scheduler, e.g.: $(0.004445 + 0.001506) * 5 = 0.02978 \approx 0.033096$, it is important to mention that the public input gate is more efficient than a normal secret input gate.

5 parties:	Smult call	constant multiplication gate
Time 100 000 wires	0.013967	0.066370
Time 10 000 wires	0.001344	0.006873
Time 1 000 wires	0.000127	0.000734
CPU Cycles Median	360	1733
CPU Cycles Average	393	1872

Table 6.6: Benchmarks for a constant multiplication gate by 5 parties.

Table 6.6 displays the results of repeatedly executing a simple multiplication gate 100 000, 10,000, and 1000 times. The times displayed for the input gate represent the total amount of time spent in computation by all five parties. Besides noting the low overhead of the scheduler, e.g.: $0.013967 * 5 = 0.069835 \approx 0.066370$, due to much simpler computations, this gate is significantly faster to execute than a normal multiplication gate, even ignoring the scheduler overhead.

6.5.2 Maurer with boolean circuits

The next set of benchmarks were focused on a local scenario with a wide-party scheduler to analyze the Maurer Boolean circuit implementation. In order to assess the impact of the change in finite fields, the tests were carried out in the same manner and with similar scheduler code as in the previous section.

5 parties (boolean):	add call	XOR gate
Time 100 000 wires	0.002737	0.012307
Time 10 000 wires	0.000337	0.001119
Time 1 000 wires	0.000025	0.000110
CPU Cycles Median	65	317
CPU Cycles Average	74	346

Table 6.7: Benchmarks for an XOR gate by 5 parties.

Table 6.7 refers to the benchmarks performed on a XOR gate, the equivalent of an addition gate in the last implementation analysed. The same linearity of times between wire sizes is reflected, as will be the case in the following benchmarks. As can be seen when comparing Table 6.7 with Table 6.2 of the previous implementation, a much simpler and smaller finite field significantly impacts the efficiency of the gate.

5 parties (boolean):	input_start call	input_end call	dispatch call	input gate
Time 100 000 wires	0.003390	0.001060	0.006034	0.143794
Time 10 000 wires	0.000298	0.000100	0.000621	0.013974
Time 1 000 wires	0.000031	0.000011	0.000055	0.001354
CPU Cycles Median	83	28	149	3427
CPU Cycles Average	90	29	173	3735

Table 6.8: Benchmarks for an input gate by 5 parties in a Boolean implementation.

Table 6.8 refers to the benchmarks performed on a Input gate. When compared to the previous implementation's input gate in Table 6.3, we can see that the input_start and dispatch calls are significantly more efficient here, something that can be attributed to the messages being 4 times smaller. However, there is a significant scheduler overhead in this implementation.

5 parties (boolean):	mult_start call	mult_end call	dispatch call	AND gate
Time 100 000 wires	0.008958	0.004640	0.006034	0.545920
Time 10 000 wires	0.000853	0.000480	0.000621	0.052176
Time 1 000 wires	0.000082	0.000047	0.000055	0.005421
CPU Cycles Median	220	117	149	13695
CPU Cycles Average	252	130	173	14775

Table 6.9: Benchmarks for an AND gate by 5 parties.

Table 6.9 refers to the benchmarks performed on a AND gate, the equivalent of a multiplication gate in the last implementation analysed. When comparing with Table 6.4 of the previous implementation, a very large boost in efficiency (nearly ten times as fast in the case of the mult_start call) can once again be seen, across all calls. While the execution of the whole gate is still nearly twice as fast, it's still important to note the significant overhead.

5 parties (boolean):	const_start call	const_end call	constant gate
Time 100 000 wires	0.001767	0.001056	0.015934
Time 10 000 wires	0.000181	0.000139	0.001652
Time 1 000 wires	0.000018	0.000010	0.000179
CPU Cycles Median	45	28	485
CPU Cycles Average	49	29	482

Table 6.10: Benchmarks for a constant gate by 5 parties in a Boolean implementation.

5 parties (boolean):	Smult call	constant AND call
Time 100 000 wires	0.003469	0.011110
Time 10 000 wires	0.000227	0.001027
Time 1 000 wires	0.000021	0.000116
CPU Cycles Median	89	269
CPU Cycles Average	97	314

Table 6.11: Benchmarks for a constant AND gate by 5 parties in a Boolean implementation.

Tables 6.10 and 6.11 refer to the benchmarks performed on gates specific to public inputs. The same efficiency gains when compared with an implementation with larger finite fields can be seen as with the other gates.

6.5.3 Maurer with 3 parties

So far, the results presented have been achieved with 5 parties. Benchmarks on an MPC Maurer implementation using the zK field were performed to assess the impact of the number of parties in the code. The scheduler used was adapted from the 5 party scenario. The results were as follows:

3 parties:	add call	add gate
Time 100 000 wires	0.003339	0.008833
Time 10 000 wires	0.000282	0.000817
Time 1 000 wires	0.000027	0.000078
CPU Cycles Median	77	222
CPU Cycles Average	84	248

Table 6.12: Benchmarks for an addition gate by 3 parties.

3 parties:	input_start call	input_end call	Input gate
Time 100 000 wires	0.003659	0.002269	0.063434
Time 10 000 wires	0.000446	0.000376	0.006382
Time 1 000 wires	0.000036	0.000016	0.000684
CPU Cycles Median	97	30	1642
CPU Cycles Average	103	63	1787

Table 6.13: Benchmarks for an input gate by 3 parties.

3 parties:	mult_start call	mul_end call	multiplication gate
Time 100 000 wires	0.012393	0.005206	0.216063
Time 10 000 wires	0.001215	0.000495	0.022098
Time 1 000 wires	0.000138	0.000047	0.002094
CPU Cycles Median	319	129	5529
CPU Cycles Average	348	147	6307

Table 6.14: Benchmarks for a multiplication gate by 3 parties.

3 parties:	const_start call	const_end call	constant gate
Time 100 000 wires	0.003763	0.001215	0.016382
Time 10 000 wires	0.000473	0.000110	0.001591
Time 1 000 wires	0.000035	0.000012	0.000166
CPU Cycles Median	103	32	589
CPU Cycles Average	106	34	515

Table 6.15: Benchmarks for a constant gate by 3 parties.

3 parties:	Smult call	constant multiplication gate
Time 100 000 wires	0.005169	0.016727
Time 10 000 wires	0.000479	0.001641
Time 1 000 wires	0.000050	0.000147
CPU Cycles Median	137	434
CPU Cycles Average	146	471

Table 6.16: Benchmarks for a constant multiplication gate by 3 parties.

The number of parties, as expected, has a significant impact on the performance of the jasmin code, but there is significant scheduler overhead in the case of the gates that handle messages.

6.5.4 GMW

Benchmarks were also performed in a developed GMW implementation to compare the results obtained from the Maurer implementation with other protocols. The implementation involved five parties and yielded the following results:

5 parties (GMW):	xor call	xor gate
Time 100 000 wires	0.001357	0.006361
Time 10 000 wires	0.000134	0.000567
Time 1 000 wires	0.000011	0.000058
CPU Cycles Median	32	162
CPU Cycles Average	38	179

Table 6.17: Benchmarks for an XOR gate by 5 parties in a GMW implementation.

5 parties (GMW):	input_start call	input_end call	input gate
Time 100 000 wires	0.001710	0.001072	0.040041
Time 10 000 wires	0.000221	0.000142	0.005232
Time 1 000 wires	0.000017	0.000010	0.000491
CPU Cycles Median	42	25	1029
CPU Cycles Average	44	30	1131

Table 6.18: Benchmarks for an input gate by 5 parties in a GMW implementation.

5 parties (GMW):	and_start call	and_end call	AND gate
Time 100 000 wires	0.002857	0.001709	0.179403
Time 10 000 wires	0.000268	0.000188	0.017952
Time 1 000 wires	0.000026	0.000018	0.001834
CPU Cycles Median	72	43	4616
CPU Cycles Average	79	47	5060

Table 6.19: Benchmarks for an AND gate by 5 parties in a GMW implementation.

As can be seen, the jasmin code outperformed the Maurer Boolean circuits implementation across the board. However, there is still significant scheduler overhead, though some of that time is spent performing the Oblivious Transfer protocol.

6.5.5 Benchmark Analysis

Several findings can be drawn from the benchmark analysis. To begin, the choice of finite fields has a significant impact on performance in the Maurer implementations. Most of the time, however, this efficiency cost is unavoidable because finite fields must be tailored to the specific

application of the implementation. However, because of its importance, the finite field should be chosen with care.

Similarly, party size should be considered and minimized, as it has a significant impact on performance. Likewise, depending on where MPC is used, the party size is not very flexible.

The scheduler code that called the message creation and memory management functions was not as well optimized as the jasmin code. As a result, most of the benchmarks had significant overhead. This means that each scheduler should be tailored to the specific situation in which the MPC implementation is being integrated and thoroughly optimized to reduce overhead in message management and communication.

GMW was found to be faster than the other protocol, but it is limited to Boolean circuits. Maurer's protocol has a more complex sharing scheme but, like the MPC-in-the-head paradigm, can be adapted to a wider range of applications.

Chapter 7

Conclusion

The development of multiple MPC implementations was described in detail. The main scope of this work involved the implementation of an easily security-verifiable jasmin implementations of MPC with 3 and 5 parties aimed at being used as a component in zero knowledge proofs, which was accomplished with the Maurer's protocol implementation using the zK finite field.

As per the objectives and scope, this implementation was intended for the MPC-in-the-head paradigm. In order to accomplish this, it was integrated, verified as secure and investigated further in this context in "Machine-checked ZKP for NP-relations: Formally Verified Security Proofs and Implementations of MPC-in-the-Head"[4]. It was discovered that it performed significantly faster than another previously verified MPC protocol for the paradigm, but it performed poorly when compared to some highly optimized unverified implementations, as was somewhat expected. Nonetheless, more work can be put into the described approaches, and the potential of these technologies was demonstrated.

For efficiency, the jasmin language allowed for simple low-level assembly programming. Furthermore, the security verification proved to be simple and quick to implement, requiring only minor changes to the code to make the security proofs easier to create. This process would have been more difficult in other languages.

Furthermore, this work contributed to the expansion of Jasmin's library of cryptographic protocol implementations.

The developed implementations can also be used and adapted for other purposes by changing the finite fields, participants number or how the schedulers handle jasmin calls and memory allocation, which not only allows these implementations to be used in a wide range of situations but also allows to improve performance, as described in the analysis of the benchmarks performed on the variations of the developed implementations.

This work can also be used as a detailed guide for using, adapting, or building upon the MPC implementations created for these protocols, and it also details strategies and blueprints that can be used in the implementations of new protocols, such as message handling and memory

management.

7.1 Future Work

While the implementations were found to be efficient when compared to other fully security-verified implementations, more work can be done to approach the performance of aggressively optimized unverified implementations.

The study and application of parallel computing in these scenarios is a significant improvement that can be explored. The optimization of the execution order of circuits is an example of a unexplored area of these implementations. All circuits in the detailed implementations were executed gate by gate, and parties can spend time waiting for messages and doing nothing. With changes to the code and design, this time could be used to partially or fully compute other gates instead of being wasted, resulting in a significant performance boost.

The main focus of this work was to make the main jasmin code as efficient as possible, with a focus on message creation, memory manipulation, and properly implementing the protocols. While some attention was paid to the scheduler's memory allocation and management, the scheduler's design can be further explored and optimized, particularly in the communication and message handling parts, which cause significant overhead.

Finally, while this study concentrated on two protocols, other generic MPC protocols are widely used. The developed strategies can be used to simplify the implementation of new protocols and allow for the creation of implementations for additional scenarios.

Bibliography

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[2] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1607–1622, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 9781450367479. doi:10.1145/3319535.3363211.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982, 2020. doi:10.1109/SP40000.2020.00028.

[4] José Bacelar Almeida, Manuel Barbosa, Manuel L Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. Machine-checked zkp for np-relations: Formally verified security proofs and implementations of mpc-in-the-head. Cryptology ePrint Archive, Report 2021/1149, 2021. https://ia.cr/2021/1149.

[5] José Carlos Bacelar Almeida, Manuel Barbosa, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. Machine-checked zkp for np-relations: Formally verified security proofs and implementations of mpc-in-the-head, 2021.

[6] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for boolean and arithmetic circuits. Cryptology ePrint Archive, Report 2016/969, 2016. https://ia.cr/2016/969.

[7] Amos Beimel. Secret-sharing schemes: A survey. In *International conference on coding and cryptology*, pages 11–46. Springer, 2011.

[8] Azer Bestavros, Andrei Lapets, and Mayank Varia. User-centric distributed solutions for privacy-preserving analytics. *Communications of the ACM*, 60(2):37–39, 2017.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[10] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proc. Priv. Enhancing Technol.*, 2016(3):117–135, 2016.

[11] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *Cryptographers' Track at the RSA Conference*, pages 416–432. Springer, 2012.

[13] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America*, pages 40–58. Springer, 2015.

[14] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[15] Wenliang Du and Mikhail J. Atallah. Secure multi-party computation problems and their applications: A review and open problems. In *Proceedings of the 2001 Workshop on New Security Paradigms*, NSPW '01, page 13–22, New York, NY, USA, 2001. Association for Computing Machinery. ISBN: 1581134576. doi:10.1145/508171.508174.

[16] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3), 2017.

[17] Hugo Krawczyk. Secret sharing made short. In *Annual international cryptology conference*, pages 136–146. Springer, 1993.

[18] Ueli Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370 – 381, 2006. ISSN: 0166-218X. Coding and Cryptography. doi:https://doi.org/10.1016/j.dam.2005.03.020.

[19] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 35–52, 2018.

[20] Michael O Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptol. ePrint Arch.*, 2005(187), 2005.

[21] Dragoș Rotaru. Honeybadgermpc. https://github.com/initc3/HoneyBadgerMPC, 2020.

[22] Nikolaj Sidorenco, Sabine Oechsner, and Bas Spitters. Formal security analysis of mpc-in-the-head zero-knowledge protocols. *IACR Cryptol. ePrint Arch.*, 2021:437, 2021.

[23] Pierre-Yves Strub and Vincent Laporte. Jasmin language. https://github.com/jasmin-lang/jasmin, 2021.

[24] Pierre-Yves Strub, Francois Dupressoir, Alley Stoughton, and Shih-Han Hung. Easycrypt. https://github.com/EasyCrypt/easycrypt, 2021.

[25] Unbound Tech. How to control your own keys (cyok) in the cloud. White Paper available from https://www.unboundtech.com, 2018.

[26] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. Cryptology ePrint Archive, Report 2017/189, 2017. https://ia.cr/2017/189.

[27] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://ia.cr/2015/1153.

[28] Paul Zimmermann, Alexandre Casamayou, Nathann Cohen, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, et al. *Computational mathematics with SageMath*. SIAM, 2018.

[29] Michael Zohner, Daniel D., Lennart Braun, Thomas Schneider, and Oleksandr Tkachenko. Otextension. https://github.com/encryptogroup/OTExtension, 2020.