CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Virtual Outsourcing of Verifiable Computation with Function Hiding

Master's thesis in Computer Science and Engineering

Christian Roos

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Virtual Outsourcing of Verifiable Computation with Function Hiding

Christian Roos

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Virtual Outsourcing of Verifiable Computation with Function Hiding

Christian Roos

Supervisor 1: Carlo Brunetta, Computer Science & Engineering
Supervisor 2: Georgia Tsaloli, Computer Science & Engineering
Examiner: Alejandro Russo, Computer Science & Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Virtual Outsourcing of Verifiable Computation with Function Hiding

Christian Roos
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

For thousands of years, humans have utilized machinery to solve computational problems out of the reach of the human mind. While the problems have become more and more complex, the machines that are used to solve these problems have become smaller and smaller. This has paved the road for a concept known as *computation outsourcing* which is based on having small devices send the problem to larger machines that are better equipped to solve the problems of the users. This concept however, raises the questions of whether the solutions received in such a outsourcing process *can be trusted* or not and *how* to be able to outsource the computations without disclosing any private information.
This thesis analyses the state-of-the-art of the cryptographic primitives for verifying data received from outsourced computations and strategies for hiding the input data and functions. The thesis further looks at different solutions that combine the analysed concepts in order to create a verifiable outsourced hidden computation scheme. An implementation of such methods in the form of a python program called *SimpleUCEvaluator* which evaluates universal circuits using a Yao's garbled circuit evaluator implementation is created and the performance of the program is analysed. The result indicates that current state-of-the-art techniques are not quite efficient enough to truly be used in practice. The process of transforming a program into a universal circuit and the process of generating a garbled circuit is very computationally expensive and, since there is no efficient way of using a garbled circuit more than once, this process can not be amortized over multiple evaluations. In some cases, *SimpleUCEvaluator* is slower by a factor of ten thousand when compared to the evaluation of a native version of the program. While in some cases, where the function must be kept hidden at all costs, this might be acceptable, in most cases this is too slow to be usable.

Keywords: verifiable computation, private function evaluation, yao's garbled circuit, oblivious transfer, universal circuit, boolean circuit.

Acknowledgements

I would like to take the time to thank my two supervisors, Carlo and Georgia, for all the help and support throughout the whole process. I would also like to thank my family. I could not have done any of this without all of your help.

Christian Roos, Gothenburg, February 2021

Contents

List of Figures

List of Tables

1

Introduction

For thousands of years, mankind has utilized different objects to solve scientific problems. Such an object, dating back to ancient Greece, was found in the early 20th century in the sea surrounding the greek island Antikyhera. Named after the location it was found, the Antikythera Mechanism consisted of a number of cog-wheels powered by a rotating lever. It is believed that the mechanism was used to predict astronomical stars' positions and cosmic events. Thus being generally considered to be the first analog computer. In contrast to this ancient machinery, whose sole purpose was to solve a **single** problem, modern computers are generally constructed to solve a large class of various problems, from calculating complex mathematical functions to communicating with people from all over the globe.

In its simplest form, an evaluation consists of a problem to be solved and the user's provided data, required to solve the problem. The computer, using the supplied data, solves the problem and the result of the solution is delivered back to the user. This is the basic concept of the evaluation: solving a problem using external data and presenting the result of the solved problem back. The basic process involves a **single** machine delivering solutions to a **single** user but in reality, it can be much more complex.

In the modern world, systems are built using a large number of computers that respond to queries from multiple users at once. In addition, the problems that need to be solved are more complex and thus require more computational power. Where there once was a single computer solving a simple problem once a day, has become a cluster of interconnected computers synchronously computing the solutions of increasingly complex problems for thousands of users every second. This progress has paved the road for the idea of *cloud computing*, which is moving data storage and problem solving from local devices, to remote specialized computers. The user supplies the local device, which now has the task of being the interface between the problem solver and the user, with all the data required to solve the problem. The local device then shares this data with an available remote device which solves the problem. The solution is then sent back through the system to the original user as illustrated in Figure 1.1. This is the essence of cloud computing.

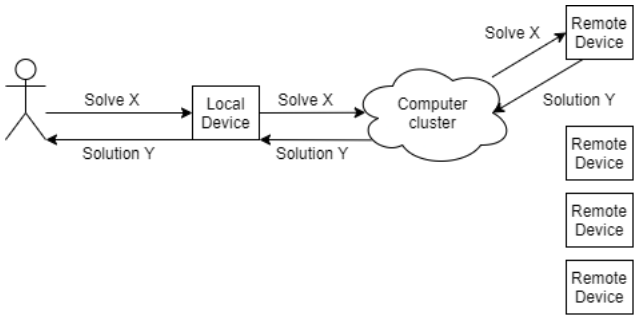


Figure 1.1: The process of cloud computing.

Cloud computing is just one implementation in a much broader concept commonly known as *outsourcing computation*. Another concept which utilizes remote devices to do the heavy lifting is called *crowd computing*. Unlike cloud computing, crowd computing makes use of one centralized server that both provides the problem and the needed data to solve it. The server then divides the problem into smaller parts and distributes these partitions of the computation to a network of remote devices. Each device in the network solves their part of the problem and returns their solution to the centralized server. Using all partitions of the solution the server proceeds to piece them back together to find the solution for the original problem.

The main difference between cloud and crowd computing is that the latter trades security and correctness for flexibility, since any device can contribute to solving the problem [1] [2]. The resources, *i.e.* the remote devices, utilized by cloud computing on the other hand, are typically more established and controlled. [3] [4].

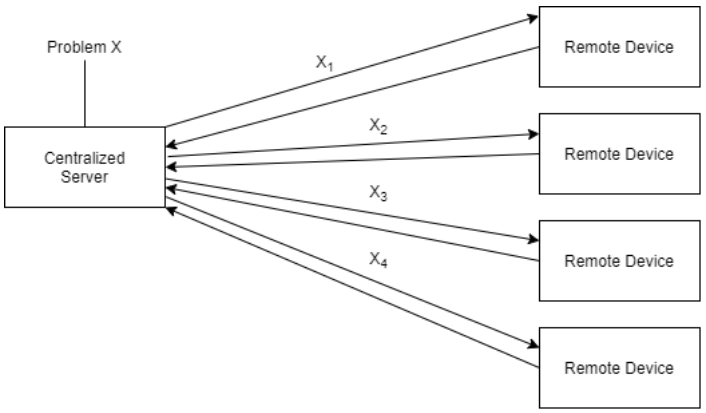


Figure 1.2: Crowd computing using one centralized server and four remote devices.

1.1 Outsourced Verifiable Computations

One major weakness shared by both cloud computing and crowd computing is the fact that, by introducing actors that are potentially unknown to the user in the process, there are no guarantees that the solutions are indeed correct. In the basic computation process there are only two actors, the user and the device solving problems. By accepting the solution and trusting it to be correct, the user irrevocably

trusts the device that provided it. Most of the times the devices used to solve the problems come from trusted manufacturers. However, when the problem is shared with remote devices operated by a third party the solutions become unreliable, especially since the user might not necessarily know exactly **which** device is actually performing the computation. This raises the question of **how** a received solution from an outsourced computation can be made trustworthy. The servers that perform the evaluation, known as *evaluators*, could provide faulty solutions to sabotage an evaluation, *i.e.* computation, and, without any security measures, there would be no way for the user to know whether the solution they have received is correct or not.

To handle this problem, the concept of *verifiable computation* can be used. Verifiable computation is the concept of verifying a solution or output of an evaluation **without** having to evaluate the function itself. Since the main idea of outsourcing computations is to allow a weak device to solve a complex problem, it is important that a potential verification process is **significantly less** costly in terms of, for example, memory usage or computation time, than solving the problem. Intuitively, if this requirement is not met, the weaker device might as well solve the problem itself.

The most common method for verifying a computation is for the evaluator to generate some sort of **proof** along with the result of an evaluation. This proof commonly consists of additional data that can be analyzed by the user to verify the correctness of the result. Figure 1.3 provides an example of verifiable computation in the context of outsourced computation. In this example, a function $F(X)$ together with some input x_1 is sent to the evaluator. The evaluator solves the function and uses a proof generating algorithm to get a proof of evaluation. Both the proof p and the output y is then sent back to the user, who utilizes a verification algorithm to verify the correctness of the evaluation. This, then concludes the process.

The main differences between a basic outsourced computation, such as cloud computing, and a verifiable outsourced computation is the addition of the verification and the proof generation algorithms that allow the user to verify the proof provided by the evaluator. The user only accepts the output if the corresponding evaluation proof is accepted by the verification algorithm. If an evaluator decides to provide an incorrect proof, *i.e.* a proof that does not pass the verification or no proof at all, the user will simply not accept the solution. This is the core principle of a basic *verifiable computation scheme*. There exist in literature, more complex variants that provide additional properties such as keeping the input hidden from the evaluator, or aims to complete the basic verification scheme in more efficient ways.

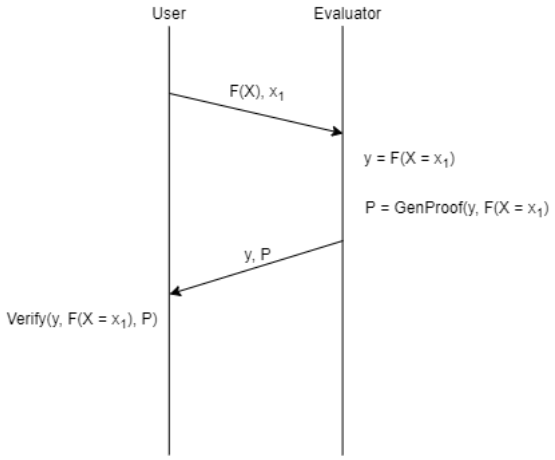


Figure 1.3: Communication overview of outsourcing function $F(X)$ over input x_1.

One thing to note in the example of Figure 1.3 is that **both** the input **and** the function itself are shared with the evaluator. This might present a problem in certain cases where either the input, the function, or both contain sensitive information that the user would prefer not to share with an unknown third party evaluator. In these cases, the basic verification protocol must be altered to include some sort of *privacy preserving mechanism* to allow the user to outsource the computation without disclosing any secret information. One approach to solve this problem is to utilize a modified version of *secure function evaluation* called *private function evaluation*. Typically, secure function evaluation protocols allow two users to evaluate a publicly known function using private inputs. By modifying the secure function evaluation protocol to privatize the function, a private function evaluation protocol is created. Generally, both users know the purpose of the function, *e.g.* to calculate the tax rate of an income, but any further details of the function remain hidden from the opposing party. The owner of the function might want to keep evaluation details private; for example a bank might only grant loans to people with a yearly income over a threshold. By outsourcing the private function evaluation protocol and using it in a verification process, such as the one in Figure 1.3, the verification process can adopt the additional properties of private inputs and functions from the private function evaluation protocol.

1.2 Related Work

Recently, much of the research surrounding verifiable computation protocols has been focused solely on improving protocols to better handle one type of situation; namely a scenario where verifiable computation is used to verify an evaluation performed by some external party. There are however, situations where verification properties would be useful even when the evaluations are not outsourced. In critical situations where any delay might result in serious consequences and external factors such as human error or external tampering might result in faulty calculations, access to fast verification protocols can be a determining factor. The aim of this thesis is to look into expanding current verifiable computation protocols to handle

the aforementioned scenario with the extended property of function hiding. This will be done by extending state-of-the-art protocols with private function evaluation techniques and evaluating how well the combination performs in the single device scenario and using regular hardware. In this thesis, the term "regular hardware" refers to ordinary commercially available desktop computers or laptops.

The concepts of verifiable computation and private function evaluation are both well researched areas, and there are several state-of-the-art protocols to consider. In verifiable computation these include, but are not limited to:

- *Pinocchio* [6], a non-interactive verifiable computation protocol with input privacy that supports public verification.
- *Gennaro et. al.* [5], a non-interactive protocol based on fully homomorphic encryption.
- *Allspice* [18], an interactive proof based protocol which implements a number of other VC protocols and chooses the best suited depending on the function that is evaluated.
- *Clover* [17], an interactive proof protocol which relies on multiple, non-colluding evaluators to verify proofs.
- *Buffet* [24], which is the latest protocol in a long list of implementations including *Zaatar* [19], *Pepper* [7], *Ginger* [20] and *Pantry* [26] that all belong to the *pepper project* [8].

In private function evaluation the protocols that were considered were:

- *Switching Networks* [30].
- *Universal Circuit Evaluation* [11] [12] [32].
- *Semi-Private Function Evaluation* [34].
- *Software Guard Extensions* [9].

1.3 Thesis Content Overview

Chapter 2 provides a theoretical background, the terminology and the primitives used in the different protocols that are mentioned in the theis. Chapter 3 deals with verifiable computation and provides the definition of verifiable computation and background information of the different protocols from the literature that implement the concept. In Chapter 4, the notion of private function evaluation is described in detail and, as in the previous chapter, some methodologies of interest for this thesis are presented. Chapter 5 provides the explanation of the implementation made and presents the results of several test evaluations and discusses the outcomes.

2
Preliminaries

The aim of this chapter is to provide formal definitions of the concepts and primitives that are used throughout the thesis. These include basic concepts and notations of graph theory, cryptographic primitives and methodology for alternative program representations. Furthermore, the chapter introduces the related concepts of homomorphic encryption and secure function evaluation.

2.1 Circuit Representations

A ***gate*** can be seen as a "box" that provides some sort of functionality. A gate takes a number of inputs and applies a function to generate a number of outputs. A gate can represent different classes of functionality depending on the nature of the gate, *i.e.* whether it is an arithmetic gate - which allows for arithmetic operations such as addition and multiplication - or a boolean gate, which allows for boolean operations such as AND and XOR. For example, Figure 2.1 shows an arithmetic addition gate which makes use of two inputs. The output of the gate is the functionality applied to the inputs, *i.e.* the addition of the two inputs.

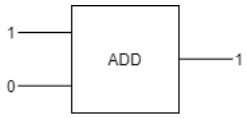


Figure 2.1: An example of a gate that adds two inputs together to get an output.

The inputs and outputs to a gate are referred to as ***wires***. For example in Figure 2.1, there are a total of three wires; the two input wires that are located to the left of the gate and represent one value each, and the third wire which is located to the right of the gate and represents the output of the gate and contains the value resulting from the addition performed by the gate.

All gate types come with its own ***truth table*** which is used to translate inputs into outputs. Thus, the truth table is basically the blueprint of the gate which maps two inputs to a certain output. Table 2.1 shows an example of the truth table of an AND gate. Each row in the truth table represents an input combination together with the correct output. Looking at row two it can be seen that the input combination $(0, 1)$ results in the output 0. This can be confirmed by looking at the properties of an AND gate. The AND gate returns one **if, and only if,** both input wires contain the value one. In this example, one wire contains the value zero and the other wire contains the value one. Therefore, the output will be zero. Using this process, the

truth table is created by checking all the combinations of inputs and printing the corresponding output. In this particular case, the only row that will return the value one is row four, which contains the input combination $(1, 1)$.

Table 2.1: The truth table of an AND gate.

Input	Output
(0,0)	0
(0,1)	0
(1,0)	0
(1,1)	1

Gates can be connected to each other by utilizing the same wires, e.g. an output wire from one gate can be used as an input wire to another. Such a concept, that consists of a collection of any number of gates that are connected to each other is referred to as a ***circuit***. In addition to gates and connecting wires, a circuit consists of ***circuit inputs*** and ***circuit outputs***. Made up of external wires, circuit inputs and circuit outputs can be considered as the start and the end, respectively, of circuits. The concept of circuits is a primitive that is commonly used to alternatively represent programs in verifiable computation and private function evaluation protocols, since circuits can be used to represent complex high level programs using only basic boolean or arithmetic operations.

For the concept to work however, there is an additional requirement related to how the gates are connected. To avoid infinitive looping, the circuit must always have a direction in which the data flows, *i.e.* no feedback loops are allowed in the circuit. This requirement limits what gates can be used with which wires. In addition to this limitation two more properties - called ***fan-in*** and ***fan-out*** - must be considered in certain types of circuits. These properties limit the number of wires that can be shared and connected to a gate. The fan-in property limits how many input wires that a gate is allowed to handle, while the fan-out property limits the number of gates or circuit outputs that are allowed to use a gate's output wire. While the number of inputs to a gate and the fan-in property are the same, it is important to distinguish between the number of outputs and the fan-out property. The number of outputs determine how many different output wires that are connected to a gate, while the fan-out property determines how many other gates that are allowed to use the value of a single output wire from the gate.

In the example circuit shown in Figure 2.2, three gates are depicted; G_1, G_2 and G_3. The first gate, G_1, has two input wires and is therefore described as a 2-fan-in gate. G_1 also has a single output wire which connects to two other gates. This is described as a 2-fan-out gate, even though the gate only has one output, since the number of gates or circuit outputs that utilizes the output is two. The two remaining gates G_2 and G_3 are both 2-fan-in, 1-fan-out gates. In addition, the Figure 2.2 circuit has four circuit input wires; W_1, W_2, W_3 and W_4, and two circuit output wires; W_6 and W_7.

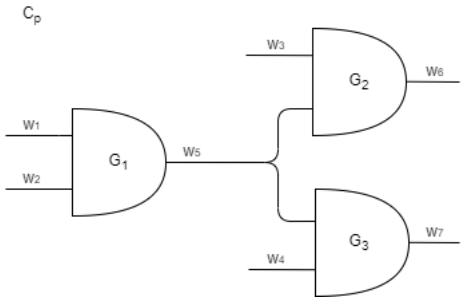


Figure 2.2: An example circuit.

In addition to ordinary circuits, this thesis also presents an extension called ***programmable circuits***. Compared to ordinary circuits, programmable circuits require additional information which is used to program the circuit to behave in a certain way. This enables a programmable circuit to return different outputs for the same input, depending on which programming that is used. The changes in behaviour can either consist of the redirection of data, e.g. by switching the values of wires, or of changing the nature of certain gates, e.g. changing a gate that adds two inputs together into a gate which multiplies the inputs instead.

2.2 Cryptographic Notions

A ***cryptographic scheme*** is used to encrypt plaintext messages into corresponding ciphertexts and back again through the use of encrypting and decrypting functions. To ensure that only intended participants will be able to use these functions on specific plaintexts and ciphertexts, a special key is utilized in the encryption and decryption processes. The cryptographic scheme therefore consists of three functions; ***KeyGen***, ***Encrypt*** and ***Decrypt***. These three functions can be implemented as follows;

- **KeyGen**$(\lambda) = k$, where k is the resulting randomized key and λ, a security parameter.
- **Encrypt**$(k, m) = c$, where k is a key received from the ***KeyGen*** function, m is some message to be encrypted and c is the ciphertext corresponding to the message m.
- **Decrypt**$(k, c) = m$, a decryption algorithm that reverses the encryption of a ciphertext. Decrypt takes a key k and a ciphertext c and returns the corresponding plaintext m.

This cryptographic scheme can be divided into two types; *symmetric* and *asymmetric* key encryption schemes. The cryptographic scheme implementation described above represents a symmetric key encryption scheme where the same key is used in both the decryption and encryption of a (plaintext, ciphertext) pair. To create an asymmetric key encryption scheme however, some minor changes to the three functions are required.

- **KeyGen**$(\lambda) = (k_e, k_d)$, where (k_e, k_d) is the resulting, randomized key pair consisting of one encryption key k_e and one decryption key k_d.
- **Encrypt**$(k_e, m) = c$, where the key from the symmetric variant is exchanged to an encryption key from the new ***KeyGen*** function.
- **Decrypt**$(k_d, c) = m$, where the key from the symmetric variant is exchanged to a decryption key from the new ***KeyGen*** function.

In this version the ***KeyGen*** function returns a key pair instead of a single key. Each of the keys are used for either the encryption or decryption of a (plaintext, ciphertext) pair. Thus, to find the ciphertext of a certain message, the decryption key from the ***KeyGen*** function must be used.

The main difference between the two versions is that in the symmetric version, anyone with access to a key is able to both decrypt **and** encrypt messages. In contrast, the asymmetric version only allows for performing one operation per key. This could for example allow a user to share the decryption key with a number of other participants while still keeping the encryption key secret. As long as the encryption key remains hidden, any ciphertext that can be decrypted using the shared decryption key must come from the user holding the encryption key, thus creating a system for signing messages.

These two versions of cryptographic schemes are used in countless applications and more complex primitives such as message signing. It is important to note though that while the base concept of the cryptographic schemes are secure, the applications and protocols utilizing these concepts might not share the same level of security. One reason for this is because the security of the system is based on keeping the key secret. An application which, intentionally or unintentionally, exposes the secret keys would no longer be considered secure. Analyzing the security of any protocol claiming cryptographic security is therefore crucial to ensure the application's integrity. Depending on the area of usage and the nature of the applications themselves, different applications require different levels of security. In order to better understand the security of applications and protocols, the behaviour of an adversary can be formalized into two different types;

- The ***semi-honest adversary*** is an adversary that, unlike the honest adversary, will attempt to retrieve any information it can while still operating within the limitations set by the protocol.
- The ***malicious adversary*** is an adversary model that is able to act in any way necessary to break the security of the given protocol. Such actions can include, but are not limited to, deviating from the intended protocol or terminating the protocol alltogether at any arbitrarily given time. When dealing with malicious adversaries it is therefore of the utmost importance to undertake the very strictest of security measurements.

There are situations in which the ability to perform operations on encrypted data is highly desirable. To enable such operations there exists a powerful primitive called ***homomorphic encryption***. In its most potent form (which is known as ***fully homomorphic encryption***) the homomorphic encryption primitive allows

anyone to "bypass" the noise added to the ciphertexts so that the resulting ciphertext is equal to the operation that was performed on the plaintexts. Other, more limited versions of homomorphic encryption, such as additively homomorphic encryption, exist as well. While the limited versions lack the ability to perform the wide range of operations that the fully homomorphic encryption primitive is capable of, they compensate with speed.

The concept of homomorphism can be formally defined as:

Definition 1 *Given an encryption scheme (KeyGen,Enc,Dec), it is said to be homomorphic if, for any pair of messages (m_1, m_2) and each generated key $k \leftarrow KeyGen(\lambda)$ and ciphertexts $c_i \leftarrow Enc(k, m_i)$, there exists an algorithm Eval that takes as input ciphertexts and outputs a new ciphertext $Eval(c_1, c_2) \rightarrow c$ such that $Dec(k, Eval(c_1, c_2)) = m_1 * m_2$.*

2.3 Alternative Program Representations

At its core, a program is nothing but a set of instructions readable to humans that tell a computer how to perform some function. The program is normally expressed through some sort of programming language such as Java or Python, which translates ordinary text into instructions that a computer can understand. Such languages have been developed in order to provide a time-saving and user-friendly alternative to writing the actual instructions for the computer. It also enables the user to more easily create and maintain complex programs.

The downside to this type of programming languages is that their user-friendliness has made them very generic and therefore ill-suited for handling tasks such as verifiable computation, which require that the user is able to write/create functions rather than depend on shortcuts/ready-made functions. This section will therefore present alternative programming languages that are based on circuit-and program constraints and subsequently better suited for use in verifiable computation and private function evaluation protocols.

The first program representation is known as *constraints* representation.

Definition 2 *A **constraint** is an arithmetic statement which represents input-and output-values and internal variables. The constraint is said to be fulfilled when a value for each internal variable is found so that the statement is true.*

Definition 3 *A **set of constraints** $C(X, Y, Z)$ is a collection of constraints with shared variables used to describe a program P where X,Y,Z are vectors representing the inputs, outputs and internal variables. The set is said to be satisfied when all constraints in the set are fulfilled and $C(X = x, Y = y, Z = z)$ gives $P(x) = y$.*

A set of constraints is generated by using a special constraint compiler which takes a program, expressed in a high level programming language. Using a special constraint evaluator, each constraint in the set is fulfilled by assigning a value for each internal variable z_i. These internal variables will contain different values depending on the values of the inputs. If there exists a combination of internal variables that fulfills all the constraints in the set, the set is said to be satisfiable.

As an example, a program which takes an integer as input and returns the input incremented with one, i.e. $y = x + 1$, is translated into a set of constraints. The result is a set containing two constraints, formally

$$C(X = x_0, Y = y_0, Z = z_0) : \begin{cases} z_0 - x_0 & = 0 \\ z_0 + 1 - y_0 & = 0 \end{cases} \tag{2.1}$$

The input is represented with x_0, the output with y_0 and there is one internal variable z_0. The constraint generation is complete and the evaluator can receive an input and iterate through the constraints to find suitable internal variable values to satisfy the set. To fulfill the first constraint, z_0 will always take the value of the input. Once the first constraint is fulfilled, the evaluator can continue to the next constraint using the newly found value of z_0 and find the value of y_0, which will be $z_0 + 1$. Since z_0 is equal to x_0, the output will always take the value $x_0 + 1$ which is equal to the original program.

The second representation is known as *universal circuits (UC)*.

Definition 4 *Let $\mathcal{C}$ be the set of all circuits of size k with m inputs and n outputs. A programmable circuit c_P is said to be a **universal circuit** if there exists a set of programmings P such that the mapping $\{c_{P_1}, c_{P_2}, ..., c_{P_{|\mathcal{C}|}}\} \rightarrow \{C_1, C_2, ..., C_{|\mathcal{C}|}\}$ is a bijection.*

UCs are special kinds of programmable circuits in the sense that they are able to simulate **any** circuit sharing the same size and the same amount of inputs and outputs. A UC is constructed in a way which allows it to take circuit descriptions as inputs. Using this description of a circuit, the UC is able to simulate the function that the circuit incorporates. This property moves the functionality that the UC provides from the universal circuit itself to the input description it takes. The main downside of using universal circuits is that they are a lot more complex than the actual circuit that is evaluated. Some of the more efficient implementations still have a complexity of $O(nlog^2n)$ [11], where n is the size of the circuit, which for larger circuits quickly becomes impractical.

In Figure 2.3 an example of how a circuit which adds two single bits into a 2-bit output is converted into a universal circuit. The universal circuit takes the description of the original circuit, P_C, which is used to program the universal circuit to mimic the behaviour of the original circuit. When the universal circuit corresponding to the original circuit is generated, the result is a universal circuit with around 80 gates [12] [13].

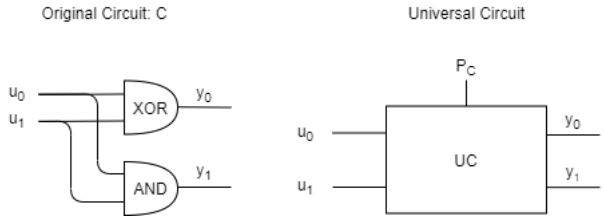


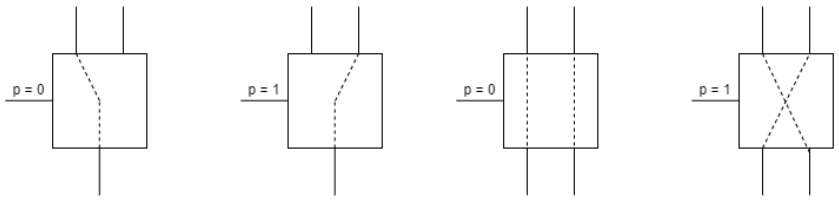
Figure 2.3: Circuit to universal circuit conversion.

Kiss and Schneider [12] build universal circuits using three special types of circuit

elements, called ***x-switches***, ***y-switches*** and ***universal gates***. The switches are used to alter the flow of inputs and data, while the universal gates are used to perform different operations on the data. What makes these elements special is the fact that they are implemented using different collections of gates. From an outside perspective, the universal circuit looks like any other circuit that is built using ordinary gates. When treated as a universal circuit however, the ordinary gates can be grouped together to create a circuit consisting only of the aforementioned special circuit elements.

The y-switch is built using two XOR gates and one AND gate and functions as a selection gate which takes two input wire values and returns one of the values unchanged. The chosen value is based on the value of the programming bit; if, for example, the programming bit is equal to zero the first wire is chosen. The functionality of a y-switch is illustrated in Figure 2.4a.

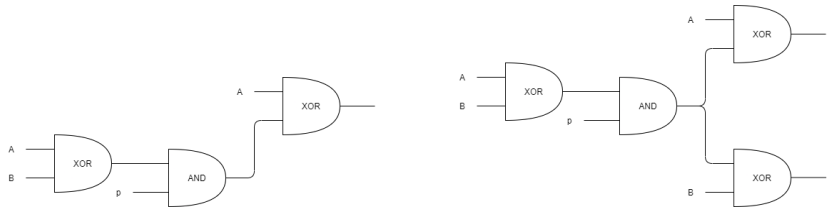
The x-switch is made of three XOR gates and one AND gate. This switch type is used to shuffle the input wires depending on the programming bit. If the programming bit is equal to zero, the inputs are returned in an unchanged order, and if the programming bit is equal to one the order of the inputs are switched. The functionality of an x-switch is illustrated in Figure 2.4b.



(a) Y-Switch options **(b)** X-Switch options

Figure 2.4: All the possible outcomes of x- and y-switches.

The circuits used to implement the switches are illustrated in Figure 2.5.



(a) Y-Switch implementation **(b)** X-Switch implementation

Figure 2.5: The implementations of the x-switches and y-switches, as used by Kiss and Schneider in [12].

The universal gate is built using three y-switches and four programming bits. These four bits can be seen as the output in the truth table of the universal gate and by using four bits, all possible truth tables of a two input gate can be simulated.

Table 2.2 displays the truth table of a universal gate and shows how the programming bits affect the output of the universal gate.

Table 2.2: The truth table of a universal gate.

Input	Output
(0,0)	p_1
(0,1)	p_2
(1,0)	p_3
(1,1)	p_4

As can be derived from the table, the programming bits make it possible to alter the output in the gate's truth table. In order to simulate, for example, an AND-gate, the programming bits $p_1 = 0, p_2 = 0, p_3 = 0, p_4 = 1$ are used. To implement this functionality, the three y-switches are used to provide the functionality described as

$$y = (p_1 \cdot \overline{A} \cdot \overline{B}) \oplus (p_2 \cdot \overline{A} \cdot B) \oplus (p_3 \cdot A \cdot \overline{B}) \oplus (p_4 \cdot A \cdot B). \tag{2.2}$$

Where the inputs are A and B, the inverse of an input is represented as $\overline{A}$, the operator "$\cdot$" represents the logical operator AND, and the operator "$\oplus$" represents the logical operator XOR.

An important thing to note when it comes to the structure of the universal gate is that the gate's programming bits are used as the inputs to the y-switches and the inputs to the gate are instead used as the programming bits for the y-switches. This is illustrated in Figure 2.6, where A and B represents the inputs.

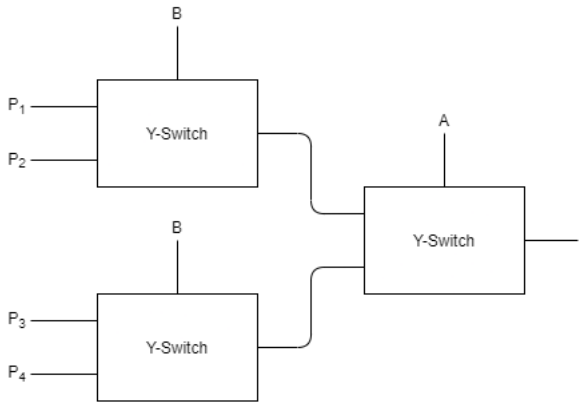


Figure 2.6: The implementation of the universal gate used by Kiss and Schneider in [12].

The third circuital representation is known as ***span program (SP)***.

Definition 5 *A **span program** is a special matrix, called span matrix, where all values are either one or zero, the values of the first column are used as selectors and the consecutive columns form row specific vectors.*

A span program consists of a span matrix divided into two parts, the selector column and the vector columns. The selector column is the first column in the

matrix and it contains a selector bit for each row. This bit decides whether the row should be active during an evaluation or not. When this column contains a one, that row is said to be active during evaluation. If the column contains a zero, the row is inactive during an evaluation. The selector column consists of input labels which corresponds to either input values or inverted input values, denoted as x_i and $\overline{x_i}$. It is therefore the input values to the span program that decides whether the rows are active or not.

When evaluating a span program, a list of inputs corresponding to each input label in the span program plus a specified target vector is required. Once the inputs have been received, the active vector set can be found by eliminating the inactive rows in the matrix. The output of the span program is found by checking whether the target vector is in the span of the active vectors. In short, this means that the sum of all vectors in the active vector set is equal to the target vector. If the target vector is in the span of the active vectors, the output of the span program is one. Otherwise, the output is zero.

By generating the correct span program and using the correct target vector, a span program $P(x)$ is said to evaluate a function $F(x)$ if and only if $P(x) = F(x)$ for all x.

Consider the example of the function F that takes two inputs and computes the AND gate and is evaluated using a span program P. The target vector chosen is $(1, 1)$, the inputs are x_1 and x_2 and the span program is built using the matrix found in Table 2.3. Using the evaluation process from before, P will be one when the sum of the active vectors are equal to the target vector $(1, 1)$. This only happens when both rows are active because neither $(1, 0)$ or $(0, 1)$ equals $(1, 1)$ by themselves but the sum $(1, 0) + (0, 1)$ does. Therefore, both x_1 and x_2 must be equal to one, which means that $P(x_1, x_2) = F(x_1, x_2)$ for all x_1 and x_2. The four different input combinations of the span program can be seen in Figure 2.7. The inactive rows for each input combination are in red and the active rows are in green.

Table 2.3: The span matrix corresponding to a two input AND operation.

x_0	1	0
x_1	0	1

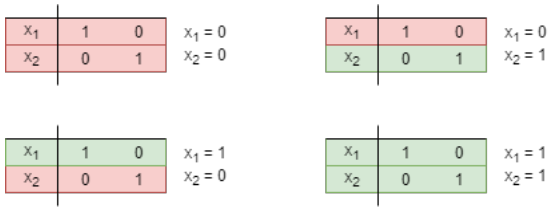


Figure 2.7: Span program of two input AND-function with all possible input combinations.

Gennaro et. al. [28] utilizes the concept of SPs when introducing a new protocol to improve certain variants of verifiable computation protocols. However, they point out some efficiency problems with SPs and propose their own version of SPs to

overcome these shortcomings called ***quadratic span programs (QSP)***. In addition, they also introduce the concept of ***quadratic arithmetic programs (QAP)***, which works similarly to QSPs, but are used to evaluate arithmetic functions instead of the boolean functions evaluated by QSPs.

The problem that is pointed out in their paper is the fact that there is currently no way to construct a truly efficient span program to an arbitrary program circuit. Therefore, any protocol based on native span programs can not be considered effective without a breakthrough in how the span programs are generated. To overcome this issue, a simpler span program is constructed that can be used to check that a given input to the program circuit is valid. The problem with this construction is that, due to the way the input checker is constructed, it allows a cheating party to assign invalid combinations to internal wires in the program circuit. To solve this problem, an additional consistency checker is constructed which makes sure that no invalid assignments are allowed.

The result is a framework which creates QSPs or QAPs, based on an arbitrary program, which are significantly more efficient than the span program which evaluates the same arbitrary program.

2.4 Secure Function Evaluation

Secure function evaluation is a generic class of protocols which allows multiple parties to jointly compute a commonly known function using private inputs. In other words, an arbitrary number of parties collaborate to evaluate a common function using private inputs from each of the involved parties. A subclass of secure function evaluation, called secure two-party evaluation, limits the number of participating parties to two. It is the secure two-party evaluation subclass that is of interest in this thesis.

As an example for the necessity of such a protocol, Yao in [15] describes a scenario where two millionaires wish to compare their wealth. In this scenario, the millionaires want to perform the comparison without disclosing any information about how much money they have. Both millionaires have their own input, their wealth, and they share a function, a comparison of two integers, and they wish to perform the evaluation without sharing their input. Yao proposed a solution to the problem which is described by Huang et. al. [16] which is based on *oblivious transfer* and *garbled circuits*.

Oblivious transfer is a special form of communication between two parties where one party has two different messages to share with the second party and the second party can choose to read one of the messages while the other remains hidden. The first party has no way of knowing which message the second part read and the second party has no way of reading both messages. This is the basic concept of what is called 1-out-of-2 oblivious transfer. In more advanced versions of the protocol, the number of sent and read messages can be increased to an arbitrary amount which extends the protocol to k-out-of-n oblivious transfer. For example an 3-out-of-5 oblivious transfer would allow the second party to choose three messages out of five sent. An implementation of 1-out-of-2 protocol is described in Protocol 1 and an illustration is shown in Figure 2.8

Protocol 1 *Party P_1 provides two messages m_0 and m_1 while party P_2 provides a decision bit c.*
1: P_1 generates a random integer a and a random generator g together with a random large prime number p. P_1 shares g and p with P_2.
2: P_2 generates a random integer b.
3: P_1 calculates $A = g^a \bmod p$ and sends A to P_2.
4: if $c = 0$, P_2 calculates $B = g^b \bmod p$, else if $c = 1$, P_2 calculates $B = A * (g^b \bmod p)$. P_2 send B to P_1.
5. P_1 creates two symmetric keys $k_0 = (B^a \bmod p)$ and $k_1 = ((B/A)^a \bmod p)$.
6. P_1 creates two cipher texts $e_0 = E(k_0, m_0)$ and $e_1 = E(k_1, m_1)$. P_1 sends e_0 and e_1 to P_2.
7. P_2 generates key $k_c = A^b \bmod p$. P_2 finds the message $m = D(k_c, e_c)$.

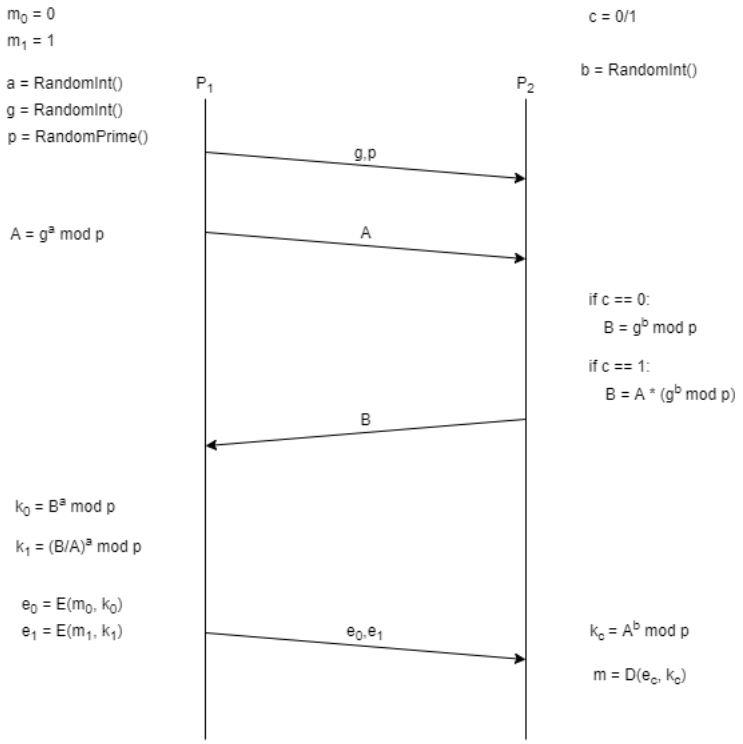


Figure 2.8: Basic implementation of 1-out-of-2 oblivious transfer protocol

The two main criteria of the oblivious transfer protocol is that P_1 can not know which message P_2 has read and that P_2 can never read more than one message. This is ensured in step 5 in Protocol 1 and more specifically on the choice of B. The value of B is either g^b or g^{a+b} and there is no way for P_1 to know which of the two that P_2 has sent. P_1 then proceeds to create keys based on the value of B in such a way that two scenarios are created:

1. P_2 sent $B = g^b$, the two keys are generated and takes the values $k_0 = g^{a \cdot b}$ and $k_1 = g^{(b-a) \cdot a}$.
2. P_2 sent $B = g^{a+b}$, the two keys are generated and takes the values $k_0 = g^{(a+b) \cdot a}$ and $k_1 = g^{((a+b)-a) \cdot a} = g^{a \cdot b}$

As can be seen from the two possible scenarios, one of the messages will always

be encrypted with the key $g^{a \cdot b}$, which is the key that P_2 is able to generate since $A^b = g^{a \cdot b}$. The message that is encrypted with this key depends on the value of B. This ensures that P_2 is only able to decrypt and read one of the messages sent by P_1 and since P_1 has no idea of which version of B that P_2 has sent, P_1 is oblivious to which message P_2 is able to decrypt.

Garbled gates are transformed boolean gates that support evaluation on secret inputs in a multi party computation setting. This is done by utilizing anonymization in the form of labeled inputs, encrypted outputs and permutation of the gate's truth table. A garbled gate is a truth table of some boolean gate, *e.g.* an AND-gate, that has been processed to hide the inputs and outputs of the gates. The process consists of transforming the truth tables in two steps. The initial state of the truth table is its original state *e.g.* an AND gate. The first step of the process transforms this basic truth table into an encrypted version of the table where all the inputs are represented using labels, made of random strings, and the outputs are the same type of labels, encrypted using symmetric keys derived from combinations of input labels. Each input label maps to the value of the value it replaces and this mapping is recorded by the garbler and kept secret. The result of the first step in the process is an anonymized version of the original truth table. The final step is the process of randomly permuting the rows of the truth table, *i.e.* randomly shuffle the rows around in the table creating a new table with the same labels as before, only in a different order.

An example of this garbling process can be seen in Table 2.4. The first table from the left is the initial truth table, the following table is the encrypted version using labels as inputs and the last table is a permuted version of the second table. The notation $Enc_{A,B}(C)$ is short for the following encryption $Enc(A, Enc(B, C))$, where the label C is first encrypted with the key B and then the result of that encryption is encrypted once more, using the second key A.

Table 2.4: The process of creating a garbled gate.

$(a,\ b)$	c
(0,0)	0
(0,1)	0
(1,0)	0
(1,1)	1

$\rightarrow$

$(a,\ b)$	c
(x_0^a, x_0^b)	$Enc_{x_0^a, x_0^b}(x_0^c)$
(x_0^a, x_1^b)	$Enc_{x_0^a, x_1^b}(x_0^c)$
(x_1^a, x_0^b)	$Enc_{x_1^a, x_0^b}(x_0^c)$
(x_1^a, x_1^b)	$Enc_{x_1^a, x_1^b}(x_1^c)$

$\rightarrow$

$(a,\ b)$	c
(x_1^a, x_0^b)	$Enc_{x_1^a, x_0^b}(x_0^c)$
(x_1^a, x_1^b)	$Enc_{x_1^a, x_1^b}(x_1^c)$
(x_0^a, x_1^b)	$Enc_{x_0^a, x_1^b}(x_0^c)$
(x_0^a, x_0^b)	$Enc_{x_0^a, x_0^b}(x_0^c)$

To evaluate the garbled gate, the garbler provides the gate's truth table and the input labels corresponding to the garbler's input values to the evaluator. Using these labels, the evaluator can create the symmetric key that is able to decrypt the output label corresponding to the inputs.

The first step, the labeling of input and encryption of output is performed to hide the inputs to a gate. Each input is replaced with a random string that reveals nothing about the value it hides. The garbler has the correct mapping which reveals that a certain label corresponds to a certain input value but to anyone else, the label is just a random string. However, this is not enough to ensure the secrecy of the inputs. The output of the table can still leak information about the input values.

For example, since the truth table is known to anyone that is going to evaluate the gate, the evaluator can easily find out whether the gate is an AND gate by looking at the output values. If the output of that AND gate is one, the evaluator knows that both the inputs are one since this is the only time that an AND gate returns one. To ensure that this can not happen, the outputs are also labeled. However, leaving the output labels in plaintext would allow the evaluator to lie about which value was evaluated so the outputs are also encrypted. The last issue is the fact that the truth tables are still ordered, which means that an evaluator can still guess the inputs if the type of the original gate is known. To eliminate this problem, the truth table is randomly shuffled which makes it impossible to guess the output by solely looking on which row in the table that is decrypted.

The idea of ***garbled circuits*** is to connect a collection of garbled gates into a circuit where the wires connecting the gates are represented using two labels each, one for each different value the wire can contain. By having the evaluator decrypt output labels, using already known input labels, the evaluator is able to find the input labels of consecutive gates which enables the evaluator to find even more labels. This allows the evaluator to find the labels corresponding to the circuit outputs without ever knowing a single value of intermediate wires in the circuit. The garbler can then receive these output labels and translate these into real values which can be shared with the evaluator.

The secure two-party evaluation protocol based on garbled circuits consists of one garbler, one evaluator and one commonly known circuit representing the function to be evaluated. The garbler starts off the protocol by garbling all the gates in the circuit and connecting them into a single garbled circuit. The garbler shares a list of all garbled table outputs in the circuit with the evaluator. From the evaluator's point of view, it receives a list of encrypted strings in sets of four, since each table has four outputs. The idea is for the evaluator to learn input labels and create symmetric keys using these labels to be able to decrypt the outputs of each table. However, at this point in time, the evaluator does not know any labels and is therefore unable to start the evaluation.

The next step in the process is to provide the evaluator with some input labels so that it is possible to start the evaluation. The garbler, which has its own set of inputs, shares the input labels corresponding to each of its input values with the evaluator. Since the input labels are random strings, this does not reveal anything about the actual input values from the garbler. Getting the evaluator's input labels is a little bit more complicated. The garbler has created the truth tables and is therefore in possession of all the evaluator's input labels. However, the garbler does not know the values on the labels since these will be provided by the evaluator. This means that, for each input wire corresponding to the evaluators input, the garbler has two labels in the form of strings. The garbler would like to share only one of these two labels since revealing both would allow the evaluator to unlock parts of the garbled circuit that does not correspond to the evaluator's input. The evaluator in turn does not wish to share which of the labels is chosen since this would reveal the value of the evaluator's inputs. To overcome this dilemma, the garbler and the evaluator engages in a series of 1-out-of-2 oblivious transfer rounds, one round for each of the evaluator's input which allows the evaluator to learn exactly one of the

input labels, the one corresponding to the evaluators input, without disclosing any information about which label the evaluator choose. Once all the input labels of the garbler and the evaluator are known to the evaluator, it can begin the evaluation of the garbled circuit. Using the input labels, the evaluator decrypts output labels of all the gates in the first layer, revealing new labels to use in the next layer of gates. This process continues until the evaluator has revealed all the labels corresponding to the circuit output values. These values can then be shared with the garbler which results in both parties having received the output of the commonly known circuit, without sharing their secret inputs.

3

Verifiable Computation

This chapter introduces the topic of verifiable computation, it then proceeds with the problem definition covered in this thesis and ends with the descriptions of state-of-the-art verification schemes present in literature.

Verifiable computation is the concept of being able to verify the output of a function without evaluating the function itself. There are many types of schemes that behave in different ways but the core functionality they provide is the same.

Definition 6 *Given a function F with inputs U and a verification function VC. VC validates an output $F(U) = y$ if $VC(y) = 1$ for all valid (U, y) pairs.*

The process of verifiable computation, as defined by Gennaro et al. [5], is divided into three steps; preprocessing, input preparation and distribution, and function evaluation and verification. The preprocessing step allows for the user to do any necessary one time computations before entering the "online" stage of the protocol. This step could for example include generation of the necessary cryptographic keys. Preprocessing is a common step for many verifiable computation schemes, such as *Pinocchio* [6], but not all protocols require this step, *e.g. Pepper* [7]. Since this step is considered "offline", *i.e.* a step to be executed before the communication with the other party has started, it is allowed to be both complex and time consuming. Most of the time, this is a one time only process. Therefore, the cost of this stage is spread across all computations. Whenever the computation is required to be executed many times, the cost of the preprocessing step becomes negligible. The second step, input preparation and distribution, is used by the user to prepare and distribute the necessary data that should be sent to the computation provider. In the last step, the computation provider has received all the information needed. The provider executes the function over the data and returns an encoded version of the output to the user.

The verification protocols are based on different proof schemes depending on the properties the protocols require. There are three proof schemes which are covered in this thesis; ***Interactive proof schemes***, ***Non-interactive proof schemes*** and ***Probabilistic checkable proof schemes***.

Interactive proof schemes share the property of requiring communication between the different parties during the verification step. The prover executes the outsourced function and ends up with some output. To verify that the output is indeed correct, the prover and the verifier initiates a communication phase where the prover tries to convince the verifier of the validity of the output.

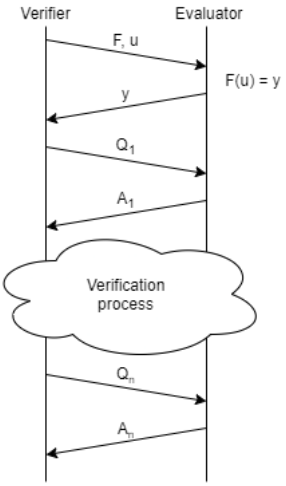


Figure 3.1: The communication process of an interactive proof scheme.

Non-interactive proof schemes have, in contrast to interactive proof schemes, the characteristic of not requiring the potentially large number of messages between the parties during the verification stage. Instead, these schemes rely on some sort of tag implementation that allows the prover to send a verification tag together with the output to the verifier. By providing all the necessary information needed to verify the output in one message, the prover does not need to convince the verifier in an online communication phase.

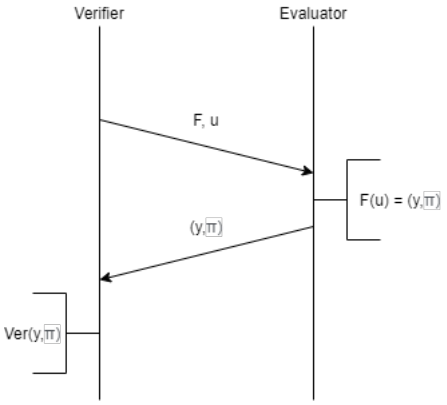


Figure 3.2: The communication process of a non-interactive proof scheme.

Probabilistic checkable proof systems can be summarized to a simple model of verification based on two properties, *completeness* and *soundness*. The properties are defined as follows.

Definition 7 *Given a function F, an input u and a valid output $F(u) = y$ with proof π, a probabilistic checkable proof system is said to be* ***complete*** *if a verifier accepts π with probability 1.*

Definition 8 *Given a function F, an input u and an invalid output y with some made up proof π, a probabilistic checkable proof system is said to be* ***sound*** *if a verifier rejects π with probability p where p describes the level of* ***soundness****.*

3.1 Problem Definition

The question this thesis tries to answer is: is it possible to evaluate and verify a secret function, in an efficient way, using current state-of-the-art protocols?

First, the problem must be defined in a more detailed way by answering the following questions:

- What does a secret function entail?
- What does **efficient** evaluation and verification mean?

To better understand these questions, consider the following scenario:

Company X is a company specialised in creating applications that are widely used by the military and most customers are governments looking to improve their national defence. The applications handle a lot of data and therefore require a fast sorting algorithm. Luckily, Company X has recently discovered a revolutionary new sorting algorithm, *Algorithm X*, which is significantly faster compared to any known algorithm. Naturally, Company X plans to include this new algorithm in their newest product. However, copying and stealing of intellectual property is very common for these kinds of products and it is not unheard of competitors that even try to sabotage any new products that hit the market. Therefore, a solution to solve these issues is of highest priority before the algorithm is released. Company X requires a solution that is able to not only hide Algorithm X from curious eyes, but also a solution that can verify the validity of the algorithm. In many cases, the application will be used in the field which often requires an offline execution mode. This means that Company X will have very limited access to the application once it has been delivered to the customer. Therefore, Company X will require a protocol that allows local verification of the output data from Algorithm X, as well as some kind of hiding mechanism for the algorithm itself. The hardware on which the algorithm will run is varying depending on the customer so the solution must be runnable on a potentially older system with limited resources.

In this scenario, a secret function would be a function which could not be copied by anyone, even by the evaluator of the function. This means that while the input to the function may be seen by the evaluator, the functionality of the program which processes the input should remain hidden even from the system evaluating it. Efficient evaluation is defined as usable in a real time application. Therefore, the verification and hiding of the functionality must be performed in a similar time frame as the original algorithm, in the eyes of the user. For example if an evaluation ran in milliseconds the same evaluation can not take more than a second to complete.

3.2 Pinocchio

Pinocchio [6] is a non-interactive protocol that generates *quadratic programs* by converting a subset of C code into circuits, which can then be translated into the corresponding quadratic programs. These quadratic programs were first introduced by Gennaro et al. [28]. In their version, the quadratic programs were based on much stronger requirements that resulted in a verification scheme that lacks efficiency. Pinocchio is instead based on quadratic programs with slightly less strict

requirements which results in a protocol significantly more efficient. The process begins by converting the chosen program into either a boolean or arithmetic circuit. Pinocchio is able to handle both types but the results show that arithmetic circuits outperform their boolean counterparts significantly. Once the program has been translated into a circuit, the result is used to create the more advanced type of program representation, a quadratic program, *i.e.* either a quadratic span program or a quadratic arithmetic program.

Before evaluation, two keys are created that are used in the evaluation and verification process respectively. The evaluation key is used as an additional input to the evaluation which assists in generating the proof π. The evaluation consists of solving a special function which involves evaluating polynomials. The proof is then generated to be a specific computable polynomial. The verification process involves using the proof and the verification key in a series of tests. Since the verification key does not reveal any information about the evaluation itself, the key can be shared to anyone to allow for public verification of the output.

The quadratic program is used to generate an evaluation key and a verification key. The keys are created using the generator of the previously chosen bilinear map in combination with the polynomial sets from the quadratic program and some random values. Once the evaluation key and the verification key has been generated, the worker can evaluate the circuit using some input received from the user. Once the circuit has been evaluated, the worker will have access to the values of each wire in the circuit. These values are used for example to replace the values of a_k and b_k of the quadratic span program in the boolean circuit version, which allows the worker to solve $p(x) = h(x) * t(x)$ and generate the proof. The verifier can then use the verification key to perform a series of tests using the output, input, proof and verification key where one of the tests is to check the quadratic program's divisibility property.

Pinocchio provides a function general verification scheme that, using the technique of generating a separate proof tag, does not require any interaction between the prover and the verifier once the keys have been shared, except for a one time sharing of the output and proof tag. In addition, this particular tag scheme allows the prover to convince any verifier that the output is a correct representation of the evaluation of some function over some input. The authors also provide a cheap extension to provide zero-knowledge proof by introducing randomization in the protocol. One major advantage with this protocol is that it solely relies on computational hardness assumptions and nothing else. The scheme provides a near practical solution where, for some functions, the scheme runs efficiently and the verification is less costly than the evaluation itself.

3.3 Allspice

The Allspice protocol [18] provides a protocol which implements, and in some cases modifies, three other protocols called *Zaatar* [19], *ginger* [20] and *CMT*[1] [21]. All-

[1]Abbreviation derived from the family names of the authors, G. Cormode, M Mitzenmacher and J. Thaler

spice handles programs in *SFDL* [22], which is a programming language specifically created for secure two party communication. SFDL resembles C or Pascal. Included in Allspice is a code analyser which takes a program in SFDL and selects the most suited protocol between the three Allspice implements.

Before going into detail of the protocols that Allspice implements, the program language SFDL must be described. This is to highlight the level of generality that Allspice can handle when it comes to the kind of programs that Allspice is able to evaluate. SFDL is a basic programming language which can handle some simple data types such as booleans and integers and can handle conditional expressions (**if**/**else**) and **for** loops over ranges that must be defined during compilation. [2]

The first protocol that is implemented in Allspice, ***CMT*** [21], is a protocol that relies on an interactive and recursive verification model. The program to be evaluated is converted into an arithmetic circuit and divided into layers of depth, *i.e.* all gates in depth 1 (input gates) are collected into one layer. Once the evaluation is complete, the verifier queries the evaluator for proof of validity of the output of the final layer. The evaluator responds with a statement that is based on the output of the previous layer of the circuit. Since the proof is based on the output of the previous layer, the verifier proceeds to query the evaluator for proof of this output. This recursive proof model is continued until the first layer is reached. At this point the verifier, who should have access to the input can evaluate the first layer and verify that the output is correct. If the output passes the verification, the recursive list of proofs can now be verified one by one since each layer of proof can be used to verify the next layer until the final output is reached. Allspice improves the CMT protocol by creating a batchable "slim" version which alters the structure of the circuit used in the original CMT protocol. CMT in its basic form takes the original circuit and adds elements to create three different parts of the circuit; the computation circuit, the propagation circuit and the comparison circuit. The computation circuit is the original circuit which just computes the original arithmetic circuit. The propagation circuit is a circuit of equal depth that is evaluated parallel to the original circuit which propagates the negation of the outputs of each gate in the computation circuit. The comparison circuit takes the output from the both other circuits and is programmed to return zero if the circuit has been evaluated correctly. The CMT-slim version used in Allspice alters the algorithm slightly making the additional circuits unnecessary and therefore removable. In addition to this change, the authors of Allspice also introduce batching which, in short, means that multiple evaluations using different inputs are put together into batches of execution.

The second and the third protocol,***Zaatar*** [19] and ***Ginger*** [20], are both based on the same concept. They use constraints to represent the input program and perform verification by checking the constraints for satisfiability. The protocols take a program expressed in SFDL and compile the program into sets of constraints. Since the input is a program in SFDL, the protocols are quite limited when it comes to what can be evaluated. Ginger however, extends the SFDL compiler to handle floating point numbers in addition to the basic types. The protocols then proceed to combine the primitives probabilistic checkable proofs and additively homomorphic encryption to generate proof for the evaluation of the constraints. Both protocols

[2] A more detailed specification is available at [44].

utilize the same method but use different versions of probabilistic checkable proofs, Ginger is based on a protocol by Arora et al. [23] while Zaatar is based on a protocol based on quadratic arithmetic programs.

Allspice implements all three of the protocols above and also implements a selector that chooses the most suitable protocol to use depending on the input program. The tool for selection generates the corresponding constraints for Ginger and Zaatar and the arithmetic circuits for CMT and tries to predict the costs for each of the protocols. The tool then selects the most suitable protocol for the given program. This comes with the upside that the most efficient protocol for a given program is chosen but it also comes with the downside of having to run the preprocessing for all three protocols to be able to make the comparison.

3.4 Buffet

The protocol called *Buffet* [24] is based on three protocols; Pantry [26], BCTV [25] and Pinocchio [6]. The goal is to build upon these protocols to overcome some significant flaws in BCTV and Pantry and achieve a solution with high expressiveness and good performance. The protocol handles an extensive subset of C code which includes most of the C language with some minor exceptions. The authors divide the protocol into three parts. First, compilation and constraints generation. Second, evaluation of the generated constraints. Third, the convincing and verification of the output. Buffet's focus is on the two first steps, Buffet implements these steps by using the protocols of BCTV and Pantry. The main focus of the protocol is to improve these parts. The last step has not been modified. Instead, step three is directly based on the Pinocchio protocol without any major modifications.

Both ***Pantry*** [26] and ***BCTV*** [25] tries to improve on the same part of the process, namely the process of translating a program, expressed in the programming language C, into constraints and how these constraints are handled. The goal for both implementations is to extend the current concept of program to constraint compilation with RAM abstraction. RAM abstraction is a way to "simulate" memory operations, namely STORE and LOAD, to introduce the concept of stateful verification. The two protocols use different methods to achieve this RAM abstraction, which results in different pros and cons. Pantry introduces two new primitives called GetBlock and PutBlock that, when translated into their constraint form, are used as building stones to create collision-resistant hash functions. These hash functions can be used to mimic memory usage by allowing the programs to define "memory locations" in the form locations in the hash table.

BCTV, on the other hand, has an entirely different approach. Before translating the C program into constraints, BCTV introduces an intermediate phase where the C program is first translated into assembly instructions for a special simulated CPU called TinyRam [27]. This intermediate step comes with some additional requirements on the code that is being translated, such as the requirement that the program is statically bound in the number of machine steps it requires. The TinyRam instructions are then compiled into constraints where the authors make the distinction between two types of constraints: **cpu execution constraints** and **memory constraints**. In short, the instructions that are executed on the simulated

CPU are divided into memory operations and all other operations. The downside to this approach is that not only the instructions must be compiled into constraints, but the entire CPU simulation including the state of the CPU and the fetching, decoding and executing operations must be included as well. By doing this, the execution of different instructions creates a transcript with the current state of the CPU and the operations that are executed. This transcript called execution-ordered transcript is used as input for the memory operations in the execution. During the CPU operation phase, each operation will be translated into a transcript where all the memory access remains "unsolved" *i.e.* the memory operations are not handled. The execution-ordered transcript is then permuted into an address-ordered transcript so that each STORE and LOAD operations can be traced, and the values on each memory location can easily be found.

The difference between the two approaches is the way that the memory operations are "paid for", *i.e.* how the cost of the additional functionality is handled. Pantry's solution introduces the memory operations GetBlock and PutBlock that are only used when memory operations are performed, all other operations are compiled into constraints just as in previous compilation processes. This means that Pantry only "pays" for the memory operations when they are used, the downside being that GetBlock and PutBlock are very expensive to execute. BCTV's result is the opposite. While performing one memory operation is not much more expensive than any other operation, all operations become more expensive and must therefore "pay" for the cost of the inclusion of the memory operations.

Buffet takes both concepts and implements a hybrid which utilizes the memory-ordered transcript and permutation of BCTV and combines it with the "pay-for-access" model of Pantry. Instead of using TinyRam, Buffet simulates memory usage using arrays and pointers from C where each element in the array simulates a memory location utilizing pointers for easy access. Buffet adapts BCTV's intermediate stage between C code and constraints where pointers and arrays are translated into LOAD and STORE operations. This intermediate stage is then translated into constraints that have a similar structure to the CPU constraints of BCTV. From these constraints, the execution ordered transcript of BCTV is generated where the values of LOAD and STORE operations are unknown, just as in BCTV. The rest of the Buffet protocol mimics the BCTV solution where an address-ordered transcript is generated. One thing to note is that the transcripts of Buffet differ from the transcripts of BCTV in that they contain much less data. The result is a protocol which adopts Pantry's efficient "regular" operations, *i.e.* non-RAM operations while keeping the cost of RAM operations lower than the costs of BCTV operations.

3.5 Clover

Blumberg et al. [17] has developed a practical *Multi-Prover Interactive Proof* scheme which uses two provers, in contrast to most schemes that use one. The scheme makes the assumption that the two provers cannot collude after during the interrogations stage of the protocol. This means that, when the verifier has started querying, the two provers can no longer communicate between each other. They propose a multi-prover interactive proof scheme with the *Proof-Of-Knowledge* property to be

used on nondeterministic circuit evaluation. Their system is called *Clover*, which is an implementation of the protocol from [17] where they add a preprocessing phase for the circuit generation which was not included in the original paper. While a multi-prover interactive proof scheme could theoretically include any number of provers, the scheme proposed by Blumberg et al. [17] focus on the scenario where there are two provers and one verifier. Two properties, *completeness* and *soundness*, must be fulfilled for the scheme to be a correct multi-party interactive protocol. The completeness property entails that the verifier should accept all correct outputs from the provers with a probability of 1. Soundness entails that when receiving an incorrect output from a prover, the verifier will accept the output with a probability of some negligible error ϵ. To achieve these properties, the following protocol is proposed. The verifier starts by sending a description of a function and the input to both provers. Once all three parties have the description, they all proceed to generate an arithmetic circuit based on the description. Then both provers evaluates the circuit using the input from the verifier and produce an output and a transcript of the evaluation. The verification concept is based on the fact that if the output is falsified then no correct transcript exists. During the verification process, the verifier queries the first prover in a number of rounds for proof of existence of a correct transcript where the last query is sent to the second prover. The final query to the second prover is used to verify that both provers provides the same answer for a random query. Since the provers are non-colluding, the probability of them providing the same answer for two different incorrect transcripts is very low.

The main disadvantage of Clover is its requirement of two non-colluding provers. This concept both requires two provers to perform the evaluation and the additional requirement that there must be some way to make sure that they are not colluding. The cost of each party included in the protocol is divided into a setup cost and a cost per computation. As reported in Figure 2 of [17].

4

Private Function Evaluation

Function hiding, or private function evaluation, is the concept of computing a function or a program while leaking as little information about the program as possible. Copying and reverse engineering of software are risks of any software once it leaves the safety of the development environment. Implementing function hiding schemes could reduce the risks of this happening.

There are a number of different properties that a verification protocol can fulfill. One being *Private Function Evaluation* or *Function Hiding*. Private function evaluation is the concept of allowing two, or more, parties to communicate and evaluate a function provided by one of the parties with inputs belonging to each party without disclosing any information about the inputs, output or function itself. This means that while party P_1 provides an input i_1 and the function f and party P_2 provides input i_2, P_2 is oblivious to f and i_1 and P_1 knows nothing of i_2. This property extends *secure function evaluation*, which provides the input and output confidentiality, by adding function hiding. A lot of the research in this area has been focused on, *universal circuits* [32] [37] [11] [12] where the main focus has been on making the universal circuits less complex. Other areas that have been researched are *homomorphic encryption*, *multi-party computation* schemes [30] and special hardware that is secure against tampering [9].

4.1 Switching Network

Mohassel and Sadeghian [30] break down a program into circuits and perform private function evaluation using something called an *oblivious switching network* evaluation. In their paper, three schemes are proposed that use the protocol in slightly different ways. The first one is a multi party scheme where any number of participants are allowed to be a part of the scheme and provide inputs for the network. The second one is a two party scheme which limits the number of participants. Both of these schemes are applied on boolean circuit representations. The third scheme however, is a two party scheme that is based on arithmetic circuits instead and additively homomorphic encryption. The one of interest for this paper is the scheme that focuses on two party communication with arithmetic circuits.

The authors divide the problem of hiding functions into two parts, hiding the **topology** of the circuit and hiding the **functionality** of the gates in the circuit. As private function evaluation is an extension of secure function evaluation, these two properties are an extension to the confidentiality that secure function evaluation already provides. The first property, topology hiding is achieved by using an

alternative way of representing the topology. The topology of a circuit is described by mapping the outgoing wires to ingoing wires $\pi_C : \{1...|OW|\} \rightarrow \{1...|IW|\}$, where OW is a combination of all input wires and all non output gate output wires and IW is the combination of all input wires. This creates a mapping where all outgoing wires have at least one connecting to an ingoing wire. The outgoing wires from output gates are excluded since output gates do not connect to any ingoing wires.

Protocol Overview. The authors assume that the number of gates, the number of input wires and the number of output wires are publicly known. Apart from this information, nothing else is disclosed. The private function evaluation scheme, divided into two parts, is described as follows.

The first part, topology hiding, includes a setup phase and two types of queries, $OMAP(x, j)$ and $Reveal(j)$. In the setup phase, P_0 generates the topology mapping representation of the circuit. P_0 also generates one random blinding vector t_i for each party i involved in the evaluation. t_i consists of random values over a range of two times the number of gates in the circuit $t_i = (t_i[1], ..., t_i[2g])$. All other parties generate random key vectors k_i with the same range as t_i. This concludes the setup phase. The rest of the protocol for topology hiding is performed in an online phase where parties send queries to P_0. The $OMAP$ queries allows parties to share their secret knowledge of an ingoing wire. P_i, where $i \neq 1$ sends the vector x_i blinded by its random vector k_i to P_0 who can then calculate the output $Out(i, l) = x_i \oplus k_i[l] \oplus t_i[l]$ where l is the output from the mapping corresponding to wire j. Any P_i can then use the query *Reveal* and provide a wire j to receive the blinded output of said wire. Since the output of the wire is blinded by both k_i and t_i, P_i can retrieve the blinded output $x_i \oplus t_i[j]$ by XORing the output of *Reveal* with its random key value $k_i[j]$. In this example, the blinding operator is XOR. However, depending on which scheme this is applied to, any blinding operator, such as modular addition or homomorphic addition, can replace XOR. The idea of this protocol is the same for all three protocols proposed in [30]. However, they are implemented in slightly different ways to better suit the actual protocols.

The second part, hiding the functionality of the gates, is achieved using a scheme named private gate evaluation. The basic idea is to allow for evaluation of a gate without disclosing the operation that was actually performed. The three different protocols proposed by Hoassel and Sadeghian [30] implement different versions of this step. Now that these two concepts, topology hiding and gate hiding, have been explained, the protocol of interest for this thesis will be described below.

Arithmetic Circuit Protocol. This protocol utilizes homomorphic encryption and replaces the boolean circuits from the two other protocols with arithmetic circuits. The protocol handles two types of gates, addition and multiplication. The topology hiding is realised using a variant of oblivious switching network evaluation. It also has a special variant of private function evaluation.

Using permutation networks a topology hiding protocol is created. The primitive used to achieve this is called *oblivious extended permutation* and in this case more specifically, *homomorphic encryption oblivious extended permutation* (*HE-OEP*). In this version, the party that has the input vector generates a key pair and encrypts the input before sending it to the other party. Using the homomorphic properties of

the ciphertexts, the second party applies the permutation network to the data and then the output is blinded using the blinding vector. The second party then sends the output to the first party. The first party can then decrypt the output to retrieve the blinded output.

The private gate evaluation protocol works as follows. One party P_0 provides the gate and some inputs u_0 and x_0. Another party P_1 provides inputs u_1 and x_1. The goal is for both parties to obtain their shares of the output c_i. First, P_1 generates a key pair sk, pk, exclusive for the gate. P_1 then uses the additively homomorphic encryption scheme to encrypt three values; $Enc_{pk}(u_1)$, $Enc_{pk}(x_1)$ and $Enc_{pk}(u_1 * x_1)$. These values are sent to P_0. P_0, knowing the gate functionality, performs some evaluation depending on the gate operation i.e. if the gate operation is addition then one evaluation is performed and another evaluation is performed if the gate is a multiplication operation. The result is two values no matter what gate operation, an output c_0 in plaintext which is P_0's share and an output $Enc_{pk}(c_1)$ which is P_1's encrypted share. The encrypted share is computed using homomorphic operations and the actual values from P_1 are never revealed to P_0. The three values provided by P_1 allows P_0 to evaluate both the addition gate and the multiplication gate in such a way that P_1 is unable to distinguish between multiplication or addition output.

The private gate evaluation together with HE-OEP are used to achieve a secure two party private function evaluation scheme, proved by Mohassel and Sadeghian [30].

4.2 Hardware

The microprocessors manufacturer Intel [31], have created a protected way to access memory and run applications called *software guard extensions (SGX)*. SGX allows the user to set up so-called *enclaves* which are encrypted environments to run applications and store data. SGX is implemented on a level which allows it to hide all information within the enclaves even from the operating system on the machine. There are works where SGX is utilized to perform private function evaluation [9] where the results are efficient private function evaluation schemes. There is however, a significant downside to using this kind of solution. The whole scheme is dependent on the hardware protection that SGX provides. This means that the scheme loses generality since it only works with certain types of hardware.

4.3 Universal Circuit

There are a number of private function evaluation protocols that depend on universal circuits [11][32][33]. The reason for this is because of the function generality property that universal circuits provide. The main idea of UC is that given a universal circuit C of size m, C can simulate any circuit representation with size m. This means that the topology of the circuit and the functionality of the gates in the circuit are kept hidden from the evaluating party. Another great reason for using UC for private function evaluation is because UC can easily be used with existing secure function evaluation schemes such as [22]. This is more or less done by using a generated UC

as the circuit to be evaluated in the secure function evaluation protocol. However, there is a major downside to UC as well. The problem is that the size of the UCs becomes too large when simulating large and complex functions. Due to the increasing complexity of UC when applied to large functions, the practicality of the concept is very limited. In summary, while UC provides function generality and is able to extend well established secure function evaluation schemes, the current state of the art UCs lack efficiency making them impractical for real applications.

4.4 Semi Private Functions

While universal circuits have the property of being function general, they lack efficiency. This is due to the fact that the complexity of universal circuits grows fast for large and complex circuits. To circumvent this issue, Paus et al. [34] proposes the concept of *semi-private function evaluation*. While a universal circuit can simulate any kind of circuit with size k, Paus et al. [34] propose a new primitive called *privately programmable blocks (PPB)* that can simulate a set of functions. In reality, PPB are a subset of universal circuits. The difference is that a PPB can be much less complex than a universal circuit since the set of functions that it simulates can have a limited range. An example would be a PPB that supports addition and subtraction. In this case, there are only two functions that must be covered and therefore the resulting PPB is very small. These PPB can be connected to each other and together form a complex program. This scheme allows for hiding of the gate functionality while leaving the topology of the program public. One major upside of this scheme is that it provides a certain level of flexibility to the user when it comes to creating programs. By having a modular structure, the programmer have the choice of only hiding parts of the program to increase efficiency. In some cases, this kind of structure can be preferable to a single universal circuit when only small parts of the program is actually confidential information. To provide full function generality, if this property is required, it is possible to implement universal circuits in the framework as well.

The base for the private function evaluation protocol is Yao's garbled circuits [15]. Where an evaluator E and a garbler G works together to evaluate some function. Using Yao's protocol, E receives some garbed circuit and garbled input from G. Using these parameters, E can evaluate the non-garbled output of the circuit. In addition to the output, E also learns the topology of the circuit. Normally, this information would allow E to extract information of the functionality of the gates in the circuit. However, the semi private function concept allows G to hide the functionality in the "semi-universal" blocks, namely the PPBs. By using PPBs, G can limit the information given to E to at most a set of functionalities that a topology represents.

The framework created by Paus et al. [34] is based on the fairplay framework [22] and is called FairplaySPF. Paus et al. [34] extends the fariplay framework and implements their own program language called *Secure Programmable Block Description Language (SPBDL)*. FairplaySPF takes a program expressed in SPBDL and creates circuit representations of the program using PPB. These circuits are represented in Fairplay's circuit language, *Secure Hardware Definition Language (SHDL)*. The

process is executed as follows, one party p_0 provides a private function, expressed in SPBDL, as input in the fairplaySPF compiler. The compiler generates a SHDL circuit. p_0 now has the option to provide input values to be merged with the circuit for optimized performance. The optimized circuit is then garbled and sent to some evaluator p_1 to be used in the private evaluation protocol.

5
Methods

This chapter will be divided into two sections. The first section discusses the different protocols covered in the previous chapters and provides an explanation to which protocols are chosen for the thesis' implementation. It will also cover how the protocols are modified to work together. The second section provides a description of how the implemented protocol is used and the environment used when the evaluations are performed.

5.1 Implementation

The major problem that requires a solution is finding a protocol that provides a verifiable output while preserving the privacy of the evaluated program. The additional requirement is the focus on keeping the solution "practical", *i.e.* finding a way of providing the aforementioned properties, while maintaining an execution time that is comparable to the execution time of the evaluated program.

The solution is based on combining verifiable computation protocols, which solve the requirement of being able to verify the output of an evaluation, with private function evaluation protocols, which covers the function hiding requirement. The two concepts are compatible due to the fact that, in most cases, verifiable computation protocols tries to solve the problem of evaluating a program in such a way that the output can be verified while, in comparison, private function evaluation protocols tend to focuses on how to modify the program itself to hide it during evaluation. This creates a natural symbiosis between the concepts where the function hiding part focuses on how the program is represented while the verification part focuses on how the program is evaluated.

When comparing the options of protocols there are two properties that are more important than others, namely the active evaluation cost, *i.e.* how much slower the evaluation phase of the protocol is compared to the native evaluation time, and how compatible the protocol is with the other type of protocols, *i.e.* the additional cost that comes with combining a private function evaluation protocol with a verifiable computation protocol.

Pinocchio can handle an extensive class of programs, provides non-interactive proof which anyone can verify and the verification is relatively inexpensive. However, the downside is the expensive proof generation step which slows down the evaluation.

Geppetto, which is built as an extension of Pinocchio, improves this aspect but unfortunately makes the program evaluation more expensive.

Clover is built on an interactive proof scheme which requires two independent

provers, i.e. proof generating parties. In the considered scenario, evaluating, generating proof and verifying on the same machine, the introduction of another party is undesirable.

Buffet introduces stateful evaluation to the verification scheme with the use of RAM to be able to handle an even larger class of programs than Pinocchio. The downside is that, even though Buffet offers an improved version, it is still based on the Pinocchio protocol and most downsides which come with it.

Allspice is a flexible protocol which utilizes three verifiable computation protocols; Zaatar, Ginger and CMT, and uses its code analysis tool to choose the best suited protocol for any given program. It covers a limited class of programs due to only handling programs expressed in SFDL and it has an expensive setup phase mainly because the setup for three protocols must be performed for each program.

Yao's garbled circuit protocol can be evaluated in such a way that the verification of the output comes with no cost. The main downside with this protocol is the expensive setup phase. To maintain the privacy of the circuit, a new garbled circuit must be generated for every new evaluation. This means that the setup cost can not be amortized over many evaluations. The greatest strength of this protocol, when considering this thesis' scenario, is its compatibility with universal circuits and semi private functions. The garbled circuit protocol can be used, without any significant modifications, to evaluate universal circuits or semi private functions and therefore obtain the property of function hiding.

The switching network protocol works for both arithmetic circuits and boolean circuits and can handle any number of participants in the protocol. Unfortunately, these properties are not too important in the considered scenario. The additional downside is the complexity of the protocol, a very complex protocol is undesirable since the protocol must be implemented together with a verification protocol.

Secure guard extensions provide a solid and functionality general way of providing the privacy property, however this creates a limitation on the types of machines that are able to run the protocol. Since these secure guard extensions only exist for certain processors, it would be impossible to use the protocol on machines which use incompatible processors.

Universal circuits and semi private functions are based on the same concept, hiding the program by creating a program which can simulate multiple other programs. Universal circuits focus on security by creating programs that can simulate a very large number of other programs. The cost for this property is that the size of the universal program is large which makes the evaluation very slow. Semi private functions trades security for speed by creating programs which can simulate a limited number of other programs. This makes it easier for an adversary to guess the nature of the program, in comparison to universal circuits, but it results in faster evaluation.

A summarization of the protocols and the pros and cons can be found in Tables 5.1 and 5.2.

Table 5.1: Comparison of verifiable computation protocols.

Protocol	Pros	Cons
Pinocchio	Non-interactive Moderate function generality No cryptographic operations Public verification Zero knowledge	Expensive proof generation Expensive setup phase
Allspice	Flexible	Requires programs expressed in SFDL Expensive setup Interactive proof
Buffet	Strong function generality Efficient evaluation and verifying	Costly setup phase
Clover	Strong security	Requires two non-colluding provers Interactive proof
Geppetto	Improved proof generation	Expensive evaluation
Yao's Garbled Circuits	Function generality Low evaluation cost Universal circuit compatibility Low verification cost	One time use setup

Table 5.2: Comparison of private function evaluation protocols.

Protocol	Pros	Cons
Switching Network	Handles both arithmetic and boolean circuits Multi party evaluation	Interactive evaluation
Secure Guard Extensions	Function generality Low overhead	Requires special hardware
Universal Circuits	Function generality Strong security	Ill suited for large programs
Semi Private Functions	Function generality Relatively efficient	Relaxed security model

The powerful primitive homomorphic encryption, which evaluates functions over encrypted data, has found its use in both verifiable computation and private function evaluation. However, the current state-of-the-art is not descriptive enough, different types of partial homomorphic encryptions *e.g.* additive homomorphic encryption, or simply not efficient enough, *e.g.* fully homomorphic encryption schemes. It was therefore not included in the comparison even though it is a potential solution to the problem.

The initial idea is combining the existing verifiable computation framework called Buffet [24] with the semi private functions created by Paus et. al. [34]. In addition, evaluation of the universal circuits implemented by Kiss and Schneider [12] will be included. Both the semi private functions and the universal circuits are based on Fairplay's programming language, Secure Hardware Definition Language (SHDL), which makes it easy to create an evaluation tool which can handle both. Using the combination of universal circuits and semi private functions allows for many different variations of the same program making it possible to perform comparisons of parameters like evaluation speed and memory usage. From this point on, the term

universal circuit will be a collective concept that includes pure universal circuits and also semi private functions hybrids.

The concept of using universal circuits in combination with verifiable computation protocols is not new. A protocol of verifiable computation on encrypted data is proposed by Fiore et. al. [37]. Their protocol relies on homomorphic encryption to perform verification and evaluation on encrypted data and they mention achieving function hiding by utilizing universal circuits. However, they apply universal circuit evaluation to another branch of verifiable computation which makes their solution different from the solution in this thesis. Another role for universal circuits in verifiable computation is proposed by Gennaro et. al. in [28]. They utilize universal circuits to reduce the preprocessing stage for the verifier. In this case, the universal circuits do not provide function hiding but instead it is used to improve preprocessing performance.

One important thing to consider is how the evaluation of universal circuits are performed. Currently, the only way to compute universal circuits without leaking the confidential parts of the circuit is to evaluate the circuit without sharing the secret programming bits between the circuit provider and the other party. Therefore, some sort of secure two party evaluation protocol is commonly used to perform the evaluation.

In this thesis, the universal circuit is evaluated using one of these protocols, namely Yao's garbled circuit protocol. The protocol allows for each party to provide their own private input which allows the party holding the universal circuit to provide the programming bits as part of its input. By doing this, the programming bits remain hidden during the evaluation due to the input privacy preserving properties of the protocol.

As it turns out, Yao's garbled circuit protocol can be evaluated in such a way that the output is both verifiable and hidden to the evaluator. Each gate in the garbled circuit takes two input labels that are used to decrypt one out of four outputs. Each label contains the potential value of a wire. Once a label has been unlocked, the corresponding wire has been "assigned" a value. Since a wire can never change value during an evaluation, it is impossible for the evaluator to find the label corresponding to the locked value of the wire. This ensures that it is in fact impossible to find more than one output label for each gate, as long as the encryption of the outputs holds. To the evaluator, these labels are just a random sequence of characters which are impossible to translate into actual output values. This property ensures that all the internal values of the circuit and the output values remain hidden to the evaluator, the labels can be used to unlock new labels in consecutive gates, but nothing else. Once the evaluator has decrypted all gates in the circuit, the circuit is said to be evaluated and all the output labels have been found.

At this point, the evaluator is unable to reveal the values behind the labels, which makes the output hidden, and since there is no way for the evaluator to find more than one output for each gate, the evaluator is unable to find output labels that does not correspond to the used input, which forces the evaluator to return correct output or return some random nonsense. This makes it easy to verify the data once the owner of the garbled circuit receives the output. If the output labels are random nonsense, i.e. does not match any of the expected output labels, the owner of the

garbled circuit can reject the output. This is possible since the owner of the garbled circuit keeps a mapping between output labels and output values, if the received output labels does not exist in the mapping, the output is invalid.

The final idea is therefore to evaluate universal circuits solely using Yao's garbled circuit protocol since adding an additional protocol to verify the output is unnecessary.

5.2 Construction

The implemented evaluation is divided into three parts, the first being a preprocessing stage performed mainly by one of the parties only, namely the verifier/program provider. The preprocessing stage is illustrated in Figure 5.1. The input to the stage is the secret program which is defined in a high level language. This program is used to generate a preliminary universal circuit and its programming. This stage has two outputs, the public universal circuit and the private universal circuit. The public version contains the circuit which will be distributed to the evaluator while the private version is the processed version of the public circuit which contains the programming.

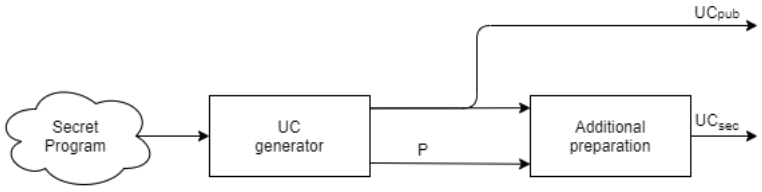


Figure 5.1: The preprocessing stage of the circuit evaluation.

The second stage is the information exchange and evaluation stage. The evaluator will receive the programming of the circuit and also receive some input from the user, as seen in Figure 5.2. However, the programming will reveal the program behind the circuit. Therefore, the verifier must provide the programming information in a way that does not reveal the original program. Once the evaluator has received all the programming bits it is able to evaluate the universal circuit. This leads to the final stage of the process; which is verifying the output. The evaluator and the verifier will exchange the necessary information so that the verifier can ensure that the output is indeed the result of the evaluation.

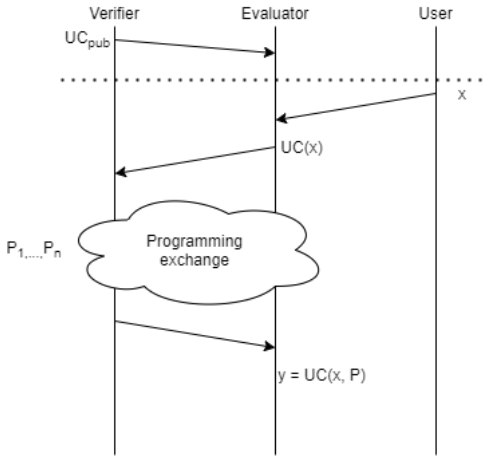


Figure 5.2: The communication and evaluation stage.

In such a system, three parties are involved in the evaluation. The verifier/program provider, the evaluator and the user. The verifier performs most of the preprocessing setup and holds all the information about the original program and the corresponding universal circuit and circuit programming. The evaluator works as an interface between the user and the circuit and performs the evaluation of the circuit. It does not provide any additional information to the process but will take information from both the verifier and the user and use it to evaluate the circuit. The user only has one task, to provide the input to the function.

The preprocessing stage has two important tasks. First, translate a program into a universal circuit and second, generate a garbled circuit from the universal circuit. The translation from program to universal circuit is done by first expressing the program in the SHDL language. Once in this form, the program can be used to generate the universal circuit by using the compiler[1] created by Kiss and Schneider [12]. When running the compiler on the SHDL program, two files containing the universal circuit and its programming are returned. The universal circuit file is used to generate a garbled circuit using the garbling process described in Section 2.4 which involves garbling the circuit gate for gate. Once this process is completed, the preprocessing stage is concluded.

The next stage is the evaluation stage, which is also divided into two parts; circuit and input sharing and circuit evaluation. The first part, circuit and input sharing, starts with the garbler sending the circuit and its inputs, in the form of labels, to the evaluator. Then, the evaluator and the garbler initiates a series of oblivious transfer rounds, one for each evaluator input, so that the evaluator can receive all the labels corresponding to its input values. Once all oblivious transfer rounds are completed, the second part of the evaluation can begin. This part consists of the evaluator decrypting output labels for each gate in the garbled circuit using the input labels received in the sharing part and any additional labels found during the evaluation. Once all gates have been decrypted, the evaluation stage is over and the final stage can be started.

The final stage, output sharing and verification, start with the evaluator sharing

[1]The compiler is available at https://github.com/encryptogroup/UC

the circuit's output labels, found in the evaluation stage, with the garbler. Once the garbler has received the output labels, it can start translating these labels into output values using a label to value mapping which is created during the preprocessing stage. Once all the output labels have been translated, the final stage is over and the entire protocol is completed.

As an example, the program shows in Listing 5.1 which is the SHDL format of the evaluation of a XOR-gate with two inputs and an AND-gate with two inputs. The first character of each line is the wire id corresponding to the line. The SHDL file is structured so that the first set of lines are all the input wires of the circuit, following are all the internal gates, then comes all the output gates and lastly there is a line with all the output wire ids listed. In this example, there are no internal gates so the output gates come directly after the inputs.

The third line, which is one of the more interesting lines of this example, contains four elements; *output gate*, *arity 2*, *table [0 1 1 0]* and *inputs [0 1]*. The first element marks the line as an output gate which means that no internal gate will use this wire. The second element shows the number of inputs to the gate while the third element shows the truth table of the gate, this example contains the truth table of a XOR-gate, and the last element shows the ids of all the inputs to the gate.

Listing 5.1: Example SHDL program.

```
0 input
1 input
2 output gate arity 2 table [ 0 1 1 0 ] inputs [ 0 1 ]
3 output gate arity 2 table [ 0 0 0 1 ] inputs [ 0 1 ]
outputs 2 3
```

The example SHDL program is used as input to the universal circuit compiler which returns two files containing the code found in Listing 5.2 and Listing 5.3.

In the universal circuit file there are five different types of lines, C, X, Y, U and O. The first line is always marked with C which represents all the input wire ids in the circuit. The lines that start with either X, Y or U represents the gate elements of the universal circuit, i.e. **X**-switches, **Y**-switches and **U**niversal gates, as described in Section 2.3, the line type is followed by two input ids and, for Y and U, a single output wire id or, for X, two output wire ids. These three gate elements all require programming data which is included in the separate programming file. The last line in the file always starts with O which is followed by all the output wire ids for the circuit.

The programming file is a list of bytes, shown as integers in the range 0-15, which represents the programming bits for each of the gate elements in the universal circuit. The first line in this file represents the programming bit for the first gate element in the universal circuit file.

Listing 5.2: Example universal circuit.

```
C 0 1
X 0 1 2 3
X 2 3 4 5
X 0 1 6 7
X 6 7 8 9
U 5 9 10
```

```
X 4 10 11 12
X 8 10 13 14
U 12 14 15
X 11 15 16 17
Y 3 17 18
Y 2 16 19
X 19 18 20 21
X 13 15 22 23
Y 7 23 24
Y 6 22 25
X 25 24 26 27
Y 27 21 28
Y 26 20 29
0 28 29
```

Listing 5.3: Example of universal circuit programming.

```
0
0
0
1
6
1
1
8
0
0
0
1
0
0
0
1
0
1
```

These two files are used to generate the garbled circuit and the input labels for the garbled circuit owner in the next step of the preprocessing stage. First the circuit garbler reads the programming file and creates all the labels corresponding to the programming bits. This is required since the programming bits will represent the input values of the garbler and must therefore be converted into labels to hide the true values of the bits. The circuit garbler then reads the universal circuit file line by line and starts creating the garbled circuit.

From the first line, the garbler creates a wire, consisting of two labels, for each input id that will later be used in the oblivious transfer stage. After the first line, each following line is translated into the circuit elements that correspond to the gate element, *e.g.* a Y-switch is translated into two XOR-gates and one AND-gate, as described in Section 2.3. In addition to the gates, two new labels are created for each new wire introduced by creating these gates. For example, the first X-switch found in the universal circuit file, found at line 2 in Listing 5.2, the two input wires are already created when the first line is parsed. The only new labels that must be generated are those for the outputs of the gate and all internal wires for that X-switch element. This process is continued until all the lines in the universal circuit

file are parsed except for the last line in the file. This line is used to collect all the output labels into a mapping of output labels and output values.

When this process is complete the garbler will have four components; a garbled circuit, the labels for all input wires, the labels for all programming bits and the mapping of output labels to output values.

A partition of a garbled circuit derived from the universal circuit in Listing 5.2 can be seen in Listing 5.4. Each line represents an encrypted label and the lines are ordered so that four lines makes up an entire truth table. When evaluating the garbled circuit, one of every four lines is decryptable while the other three can not be decrypted. In the example shown in Listing 5.4, lines 1-4 contain all the label ciphertexts of the first gate and lines 5-8 contains the label ciphertexts of the second gate.

Listing 5.4: The ciphertexts of two gates in a garbled circuit.

```
1 xbjY7Q5EnYAbXRR5HiWwkXuw+bAxsA4fmpfO4iHESKQ=
2 7AcJDk+OD2FIYD2IyI8tkVmvTuK2Bt/mgj9f/LMddoQ=
3 zajYTNZoj70M8Alta6wnSVSyS+GR32nSDCYiR3n7344=
4 5BcJr5eiHVxfzSCcvQa6SXat/LMWabgrFI6zWesi4a4=
5 gaMglz6+S/u5ya3Yvr4rD1B+bCzn3U6kn1nrKfu8Dxo=
6 VNjm9WO60Y4olJhhCq1OiTBakJntk+UeniD2/NpjT6s=
7 hUOFcEHl5ijLUvL3dwtD46zBNPKYMsBplHDQkTW0ONw=
8 641eV7PoGwjV+vi0GsLy+uN+13xKxEEflEYaE9P47nw=
```

To start off the evaluation stage, the garbled circuit and the labels of the programming bits are shared with the evaluator. The garbled circuit is formatted as a list of encrypted labels that comes in sets of four, where each set represents the output of the truth table of one gate. The labels of programming bits are labels in plaintext that can be used to decrypt the garbled circuit's encrypted labels. The next step is for the evaluator to receive the input labels corresponding to the evaluators input values. For each input the evaluator has, one oblivious transfer round is initiated where the garbler is the sender and the evaluator is the receiver. The two messages that the garbler sends are the two labels corresponding to the first input wire's id. One of the messages corresponds to the wire value 0 and the other to the wire value 1. The evaluator selects one of the values and receives the corresponding label, the unused label is never received and can therefore not be used in the evaluation (see Section 2.4). This process is performed for each input of the circuit so that the evaluator can obtain all the correct input wire labels.

Once the oblivious transfer rounds are finished, the evaluator can proceed to use the programming bit labels and the input labels to evaluate the gates in the garbled circuit one by one. Once all have been evaluated the evaluator has received all the output labels of the circuit and these are sent back to the garbler. The garbler uses its output mapping to obtain the output values corresponding to the output labels and accepts the evaluation if all the output labels can be found in the mapping.

The entire evaluation process of SimpleUCEvaluator can be seen in Figure 5.3.

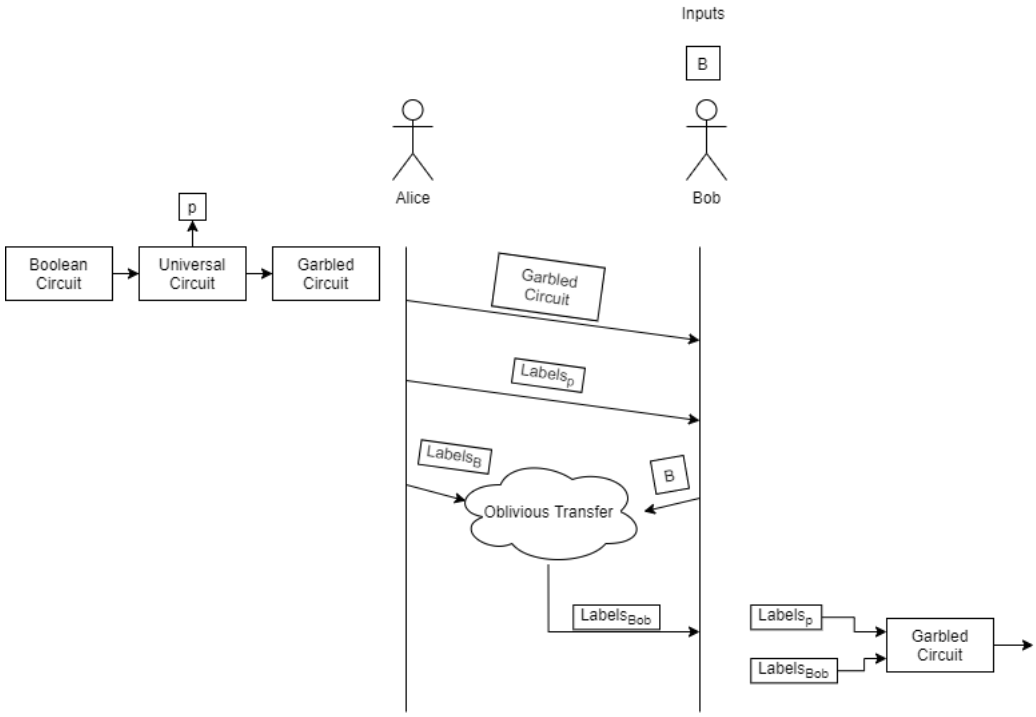


Figure 5.3: The set up and evaluation process of SimpleUCEvaluator.

The program, which implements the aforementioned protocol, is created in Python 3.8.6 and is called SimpleUCEvaluator. SimpleUCEvaluator is divided into two active parts, **alice_uc.py** and **bob_uc.py**. **alice_uc.py** takes the role of the garbler and requires the universal circuit and the programming as input. **bob_uc.py** takes the role of the evaluator and takes inputs in the form of a text file as input.

5.3 Evaluation

Three test programs expressed in SHDL are transformed into universal circuit representation of the programs and these universal circuits are then evaluated using SimpleUCEvaluator. While the evaluation and the preprocessing stage are performed, SimpleUCEvaluator measures the time it takes to complete certain steps of the process as well as the total time to complete the entire evaluation for both parties. The steps that are included in the time measurements are; preprocessing and garbling of the universal circuit; performing the oblivious transfer rounds necessary to start off the evaluation; and the evaluation of the garbled circuit. One important thing to emphasize is that the transformation from an SHDL to a universal circuit is considered an external step and therefore not included while measuring the time of the preprocessing step.

In addition to evaluating and measuring the time of the universal circuits, the evaluation and time measurement of the original program is performed as well. This is to include a clear picture of how the universal circuit transformation affects the time it takes to perform the different steps of the evaluation.

The main difference between the three programs that are evaluated is the size of the programs. The first program is a tiny program consisting of only a few gates. This

program provides insight into how usable SimpleUCEvaluator is in practice. The following two programs answer how well the evaluation scales for larger programs by significantly increasing in size, *i.e.* the amount of gates in the circuit.

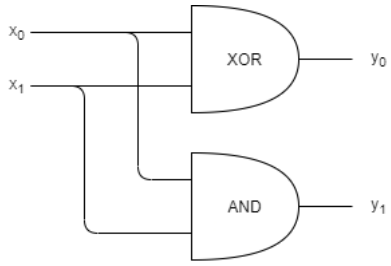


Figure 5.4: A basic two bit input adder.

The first program is a simple circuit which consists of one XOR-gate and one AND-gate giving it a size of two gates. It takes two input bits and returns two output bits. The circuit represents a simple 2-bit adder which adds the two inputs and returns a two bit output which represents the binary form of the addition. The circuit can be seen in Figure 5.4. Table 5.3 shows a comparison between the original program and the universal circuit corresponding to that program. The table compares some quantities that makes up the garbled circuit. These quantities includes; the size of the circuit, *i.e.* the number of gates; the number of wires; the number of key generations, *i.e.* the number of symmetric keys that must be generated; the number of encryptions, *i.e.* the number of plaintext to ciphertext encryptions using the symmetric keys; the number of XOR-gates, *i.e.* a partition of the total number of gates which are of the type XOR-gates; and the number of AND-gates. All these values are found by examining the circuits corresponding to the program and the universal circuit.

The number of gates, wires, XOR-gates and AND-gates are all found by simply examining the circuit and counting the occurrences of each element. The number of key generations are directly related to the number of wires in the circuit. For each wire in the circuit there are two labels, each corresponding to one of the two possible values that the wire can contain. For each label that is used to encrypt another label, a symmetric key must be generated. The number of key generations required can therefore be found by finding the set of labels that are used to encrypt other labels. This subset of all the labels in the circuit only excludes the labels that are solely used to represent the outputs to the circuit since these labels are never used to encrypt further labels. The number of encryptions are related to the number of gates in the circuit. When garbling a gate, two encryptions for each possible output of the gate are performed resulting in eight encryptions for all gates in the circuit since all gates are two input gates.

Table 5.3: Test program 1, initial comparison of circuit measurements.

Type	Size	Wires	Key Generations	Encryptions	XOR-Gates	AND-gates
Basic Circuit	2	4	4	16	1	1
Universal Circuit	76	102	200	608	54	22

The same comparison for the two remaining programs can be found in Table 5.4 and Table 5.5. These programs contain arbitrarily chosen combinations of XOR-gates and AND-gates with the purpose of examining how the performance of SimpleUCE-valuator changes for universal circuits of different magnitudes.

Table 5.4: Test program 2, initial comparison of circuit measurements.

Type	Size	Wires	Key Generations	Encryptions	XOR-Gates	AND-gates
Basic Circuit	20	40	60	160	12	8
Universal Circuit	2638	3380	6760	21104	1936	702

Table 5.5: Test program 3, initial comparison of circuit measurements.

Type	Size	Wires	Key Generations	Encryptions	XOR-Gates	AND-gates
Basic Circuit	110	130	240	880	50	60
Universal Circuit	11364	14470	28920	90912	8388	2976

When running the aforementioned programs the average time it takes to generate cryptographic keys and encrypting data is also measured to get a better understanding of how these operations affect the overall program execution time. Table 5.6 shows the result of these measurements.

Table 5.6: The average time for performing the cryptographic operations in milliseconds.

Average Key Generation(ms)	Average Encryption(ms)
0.0033	1.056

The result of evaluating the three programs is displayed in Tables 5.7, 5.8 and 5.9. The data contains five different measurements starting with the time it takes to transform a universal circuit into a garbled circuit. This is the most costly part of the whole process since this part includes generating the cryptographic keys for each label in the garbled circuit. The second measurement is the timing of the oblivious transfers performed during the online phase between the garbler and the evaluator. This timing is directly related to the number of inputs to the circuit since each input label must be sent to the evaluator using oblivious transfer. The third measurement is the evaluation timing which is a measurement of the time it takes for the evaluator to find all the labels corresponding to the input used in the evaluation. The last two measurements are the times it takes for the garbler respectively the evaluator to finish all their parts of the entire process.

Table 5.7: Test program 1, average time comparison in seconds over 500 evaluations.

Type	Garbling(s)	Oblivious Transfer(s)	Circuit Evaluation(s)	Garbler Total(s)	Evaluator Total(s)
Basic Circuit	0.066	1.14	0.0016	1.2	1.14
Universal Circuit	1.01	1.26	0.98	2.26	2.22

Table 5.8: Test program 2, average time comparison in seconds.

Type	Garbling(s)	Oblivious Transfer(s)	Circuit Evaluation(s)	Garbler Total(s)	Evaluator Total(s)
Basic Circuit	0.26	11.46	0.0051	11.73	11.47
Universal Circuit	28.65	14.41	30.56	43.06	44.97

Table 5.9: Test program 3, average time comparison in seconds.

Type	Garbling(s)	Oblivious Transfer(s)	Circuit Evaluation(s)	Garbler Total(s)	Evaluator Total(s)
Basic Circuit	1.44	11.53	0.031	12.97	11.57
Universal Circuit	128.05	11.69	128.78	139.73	140.44

6
Discussion

This chapter discusses whether it is possible to solve the problem of evaluating and verifying a secret function in an efficient way. This problem is discussed using the result of the evaluation found in Section 5.3.

Looking at the results from the three test programs, the short answer is; no it is not currently possible to solve the problem in an efficient way. However, the methods used in SimpleUCEvaluator do not reflect the potential that current state-of-the-art concepts can achieve. In fact, the implementation known as **ABY** [42] implements a far more advanced version of the same protocol and achieves much better results.

As can be observed from the result, it turns out that this thesis' implementation is a lot slower than the **ABY** evaluation. Therefore, two types of improvements of SimpleUCEvaluator will be discussed; protocol optimizations and program optimizations. One major reason why SimpleUCEvaluator can not keep up with **ABY** is due to the fact that the circuit garbler protocol that is implemented is the basic version of Yao's garbled circuits. The main cost in the protocol is the number of cryptographic operations that must be performed during the setup phase and the basic protocol has a poorly optimized implementation of the garbling process. Many improvements have been proposed that can significantly decrease the garbling time, which is the most expensive part of the setup phase and these improvements include the concepts; *Free XOR* [39] and *row reduction* and *half-and* [40].

Free XOR, as explained in the paper by Kolesnikov and Schneider [39], makes it possible to avoid encrypting the output values of XOR-gates in the circuit. Instead, the much cheaper operation of simple xor-ing the labels are used to hide the output labels. Since universal circuits are made of only XOR-gates and AND-gates where at least 2/3 of the gates are XOR-gates, Free XOR would cut out all the required key generations that are made using the input labels connected to **only** XOR-gates. Using test program 1 as an example, as seen in Table 5.3, the universal circuit consists of a total of 76 gates, where 54 are XOR-gates, and 200 required key generations. The number of input labels that are connected to only XOR-gates are 120 and when removed the result is a 60% decrease in required key generation. The additional benefit is that the number of required encryptions performed are also decreased, the amount is decreased with eight per XOR-gate in the circuit resulting in 176 required encryptions. The downside of using this concept is that more information about the circuit is leaked. While evaluating the circuit, the evaluator must know the location of all the XOR-gates because these are supposed to be evaluated in a special manner. By doing this, the evaluator must know which gates are XOR and which are not, resulting in sharing information about the circuit that should remain hidden. While this does not break the function privacy of the evaluation of

universal circuits, it still leaks more information than the basic protocol.

The **row reduction** and **half-and** methodologies both work to improve on the same parts of the protocol, namely the number of rows in each garbled table that is required. In the basic protocol, the number of garbled rows are equal to the number of rows in the truth table. In SimpleUCEvaluator, which only handles 2-fan-in gates, the number of rows is equal to four. The row reduction technique allows for a reduction to three rows per garbled table by utilizing a special function to generate the wire labels instead of relying on randomization. The half-and technique utilizes row reduction and reduces the number of rows with an additional row, resulting in a total number of two rows per garbled table. This essentially halves the amount of encryptions performed for each garbled table. Using the same example as earlier and combining the concepts of half-and and Free XOR, the number of required encryptions are decreased to a total of 88. The main downside with using the half-and concept is that the garbler must know one of the inputs to all the AND-gates in the circuit. Normally this might not be possible to guarantee however, in universal circuits, all AND-gates consist of one unknown input and one programming bit. Since all programming bits are provided by the garbler, one input will always be known allowing the garbler to utilize the half-and optimization freely.

The number of key generations and symmetric key encryptions together with the time it would take to perform these operations are displayed in Table 6.1. The times are based on the average key generation and symmetric key encryption times found in Table 5.6. The real values, found in the first row in the table, are taken from an evaluation of test program 1 using SimpleUCEvaluator, while the other values are theoretical values derived from the real values. While these theoretical values do not mirror reality exactly, they give a good indication on the improvement that can be achieved by using current state-of-the-art improvements to the garbling protocol.

There are also ways to improve the evaluation time of the program as well. The most naive form of the evaluation process forces the evaluator to try all its unlocked labels for each new gate. An improvement made in SimpleUCEvaluator is to incorporate additional data in each label so that there are two parts to the encrypted labels; the label key that can be used to decrypt further labels and a list of all the gate ids where the label is used. Using this information, the evaluator can create a map of gate ids and label keys. This allows the evaluator to know exactly which keys to use for each gate which allows the evaluator to find new labels without having to try all known labels for each gate.

Even with this improvement, the evaluator must, in the worst case, try to decrypt all four encrypted labels that correspond to a single gate. This can be avoided by using the method called *point and permute* when garbling the circuit. The idea is to randomly generate a bit value for each wire and replace the first bit in the first label with the randomly generated bit and the second label with the inverse of the randomly generated bit. Instead of randomly permuting the gate's truth table, it is instead sorted using the first bit of each of the labels. The evaluator is then able to find exactly which row in the table to decrypt by simply decrypting the row that matches the combination of the two labels used to decrypt the gate. The worst case of trying to decrypt four labels for each gate is replaced by a worst case of trying to decrypt a single label per gate which results in an improvement of up to four times

faster evaluation. Since the bits are randomly generated, the sorting of the table does not reveal any information about the values of the labels.

Aby incorporates all the aforementioned improvements and utilizes the additional improvement proposed by Bellare et. al. [43] which utilizes an optimized encryption scheme to lower the number of cryptographic operations further.

As for program optimizations, there are many improvements that can be made to better utilize the system's resources such as changing the programming language to a better suited one and utilizing threading in the setup and evaluation phases. The programming language, python, was chosen specifically due to the simplicity of the language. However, a language focused more on efficiency such as the C-family or Rust could potentially improve the runtime significantly. SimpleUCEvaluator could also be optimized as it is. Due to time limits, not much time was spent trying to improve the algorithms used and a more experienced programmer with more time could definitely improve the existing code. The second improvement, threading, was not used at all in the implementation. Both the setup phase and the evaluation phase would greatly improve if the program was rewritten to utilize multiple threads. Mainly the garbling process could benefit greatly from this since the most expensive part of the garbling is the encryption, which is currently performed in a sequential manner. Some encryption schemes support parallel encryption and/or decryption which could speed up the process significantly.

Table 6.1: Theoretical time values using the garbled circuit optimizations.

Type	Key Generations	Encryptions	Key Generation Time (s)	Encryption Time (s)
Basic Protocol	200	608	0,00066	0.64
Free XOR	80	176	0,000264	0.19
Half And	200	404	0,00066	0.43
Half And & Free XOR Combination	80	88	0,000264	0.09

Table 6.2: ABY total evaluation time in seconds.

Program	Total Time(s)
Test program 1	0.00274
Test program 2	0.0269
Test program 3	0.047

An interesting comparison is between the SimpleUCEvaluator and the Buffet framework since there are some example programs that could be compared to the universal circuit evaluation, on a general level. One good example is the matrix multiplication program which is included in the Buffet git repository. The program has quite a simple structure that takes two $m * m$ matrices as input and returns a $m * m$ matrix containing the matrix multiplication of the two input matrices. In [24], the authors evaluate this program where $m = 215$ which results in a total of 9.94 millions constraints, which require a total of 45.8 minutes to set up and evaluate. This is the evaluation time for a program iterating through a loop tens of thousands times. To put the evaluation time in context, the number of iterations in test program 3 is approximately twelve thousand times.

6.1 Conclusion

As it turns out, the evaluation of universal circuits using Yao's garbled protocol is currently not efficient enough to be used in practice. When comparing the evaluation time of the original program with the evaluation of the universal circuit, the measured time is increased by a factor of one hundred in the largest of the tested circuits (see Table 5.9). This comparison is between the evaluation of the garbled version of the original program. When the first test program is run in its native form, *i.e.* a program adding two bits together, it takes less than 0.02 milliseconds to perform the evaluation. That is faster by another factor of one hundred making the universal circuit evaluation slower by a factor of ten thousand. This can not truly be considered efficient and, even if all the improvements previously discussed were implemented, the evaluation would still be too slow to be considered efficient.

In some applications however, the trade-off between security and performance might be worthwhile. It is therefore important to note that while Yao's garbled circuit protocol, with all the state-of-the-art improvements, might not be efficient enough to solve the problem discussed in this thesis, it could still be usable in practice in certain situations.

Before the kind of evaluation tested in this thesis can be considered efficient the size of the universal circuit must be much smaller and the garbled circuit protocol must also be made much more efficient.

Bibliography

[1] https://setiathome.berkeley.edu/, accessed 2020-10-01

[2] https://foldingathome.org/home/, accessed 2020-10-01

[3] https://azure.microsoft.com/en-us/, accessed 2020-10-01

[4] https://aws.amazon.com/, accessed 2020-10-01

[5] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers," in Advances in Cryptology – CRYPTO 2010, vol. 6223, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 465–482.

[6] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly Practical Verifiable Computation," in 2013 IEEE Symposium on Security and Privacy, May 2013, pp. 238–252, doi: 10.1109/SP.2013.47.

[7] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish, "Making Argument Systems for Outsourced Computation Practical (Sometimes)," in NDSS. 2012. p. 17.

[8] https://www.pepper-project.org/, accessed 2021-01-18.

[9] O. A. Selo, M. H. Rachid, A. Shikfa, Y. Wang, and Q. Malluhi, "Private Function Evaluation Using Intel's SGX," Security and Communication Networks, Sep. 15, 2020. http://www.hindawi.com/journals/scn/2020/3042642/ (accessed Oct. 05, 2020).

[10] M. O. Rabin, "How to Exchange Secrets with Oblivious Transfer," p. 26.

[11] V. Kolesnikov and T. Schneider, "A Practical Universal Circuit Construction and Secure Evaluation of Private Functions," in Financial Cryptography and Data Security, Berlin, Heidelberg, 2008, pp. 83–97, doi: 10.1007/978-3-540-85230-8_7.

[12] Á. Kiss and T. Schneider, "Valiant's Universal Circuit is Practical" in EUROCRYPT, Austria, Vienna, 2016.

[13] http://encrypto.de/code/UC, (accessed Nov. 23, 2020).

[14] M. Karchmer and A. Wigderson, "On Span Programs," in Proceedings of the Eigth Annual Structure in Complexity Theory Conference, San Diego, CA, USA, 1993, pp. 102-111, doi: 10.1109/SCT.1993.336536.

[15] A. C. Yao, "Protocols for secure computations," in 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982), Nov. 1982, pp. 160–164, doi: 10.1109/SFCS.1982.38.

[16] Y. Huang, D. Evans, J. Katz and L. Malka "Faster secure two-party computation using garbled circuits," in USENIX Security Symposium, 2011, pp 331-335

[17] A. Blumberg, J. Thaler, V. Vu, and M. Walfish, "Verifiable computation using multiple provers," IACR Cryptol. ePrint Arch., 2014.

[18] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish, "A Hybrid Architecture for Interactive Verifiable Computation," in 2013 IEEE Symposium on Security and Privacy, May 2013, pp. 223–237, doi: 10.1109/SP.2013.48.

[19] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish, "Resolving the conflict between generality and plausibility in verified computation," 622, 2012. Accessed: Sep. 18, 2020. [Online]. Available: http://eprint.iacr.org/2012/622.

[20] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," p. 16.

[21] G. Cormode, M. Mitzenmacher, and J. Thaler, "Practical verified computation with streaming interactive proofs," in Proceedings of the 3rd Innovations in Theoretical Computer Science Conference on - ITCS '12, Cambridge, Massachusetts, 2012, pp. 90–112, doi: 10.1145/2090236.2090245.

[22] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay — A Secure Two-Party Computation System," Proceedings of the 13th conference on USENIX Security Symposium - Volume 13." https://dl.acm.org/doi/10.5555/1251375.1251395 (accessed Oct. 15, 2020).

[23] S. Arora, C. Lund, R Motwani, M. Sudan and M Szegedy, "Proof verificaiton and the hardness of approximation problems," Proceedings., 33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, PA, USA, 1992, pp. 14-23, doi: 10.1109/SFCS.1992.267823.

[24] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," p. 15.

[25] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct Non-Interactive Arguments for a von Neumann Architecture," IACR Cryptol. ePrint Arch., 2013.

[26] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, New York, NY, USA, Nov. 2013, pp. 341–357, doi: 10.1145/2517349.2522733.

[27] https://www.scipr-lab.org/doc/TinyRAM-spec-0.991.pdf, accessed 2020-10-13

[28] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic Span Programs and Succinct NIZKs without PCPs," in Advances in Cryptology – EUROCRYPT 2013, Berlin, Heidelberg, 2013, pp. 626–645, doi: 10.1007/978-3-642-38348-9_37.

[29] C. Costello et al., "Geppetto: Versatile Verifiable Computation," in 2015 IEEE Symposium on Security and Privacy, May 2015, pp. 253–270, doi: 10.1109/SP.2015.23.

[30] P. Mohassel and S. Sadeghian, "How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation," in Advances in Cryptology – EUROCRYPT 2013, Berlin, Heidelberg, 2013, pp. 557–574, doi: 10.1007/978-3-642-38348-9_33.

[31] https://www.intel.com/content/www/us/en/homepage.html, accessed 2021-01-03

[32] A.-R. Sadeghi and T. Schneider, “Generalized Universal Circuits for Secure Evaluation of Private Functions with Application to Data Classification,” in Information Security and Cryptology – ICISC 2008, Berlin, Heidelberg, 2009, pp. 336–353, doi: 10.1007/978-3-642-00730-9_21.

[33] T. Schneider, “Practical Secure Function Evaluation,” In Informatiktage - 2008 pp. 37-40.

[34] A. Paus, A.-R. Sadeghi, and T. Schneider, “Practical Secure Evaluation of Semi-private Functions,” in Applied Cryptography and Network Security, Berlin, Heidelberg, 2009, pp. 89–106, doi: 10.1007/978-3-642-01957-9_6.

[35] https://www.pepper-project.org, accessed (2020-11-23)

[36] https://www.virtualbox.org/, accessed (2020-12-08)

[37] D. Fiore, R. Gennaro, V. Pastro, “Efficiently Verifiable Computation On Encrypted Data,” in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, USA, Scottsdale, Arizona, USA, 2014, pp. 844-855, doi: 10.1145/2660267.2660366

[38] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, “Reusable Garbled Circuits and Succinct Functional Encryption,” 733, 2012. Accessed: Sep. 22, 2020. [Online]. Available: http://eprint.iacr.org/2012/733.

[39] V. Kolesnikov and T. Schneider, “Improved Garbled Circuit: Free XOR Gates and Applications.” in Automata, Languages and Programming. ICALP 2008. Lecture Notes in Computer Science, vol 5126. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-70583-3_40

[40] S. Zahur, M. Rosulek and D. Evans, “Two Halves Make a Whole” in Advances in Cryptology” - EUROCRYPT 2015. EUROCRYPT 2015. Lecture Notes in Computer Science, vol 9057. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-46803-6_8

[41] B. W. Reichardt, "Span Programs and Quantum Query Complexity: The General Adversary Bound Is Nearly Tight for Every Boolean Function," 2009 50th Annual IEEE Symposium on Foundations of Computer Science, Atlanta, GA, 2009, pp. 544-551, doi: 10.1109/FOCS.2009.55.

[42] D. Demmler, T. Schneider and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation", 2015 in Network and Distributed System Security Symposium, doi: 10.14722/ndss.2015.23113.

[43] M. Bellare, V. T. Hoang, S. Keelveedhi and P. Rogaway, "Efficient Garbling from a Fixed-Key Blockcipher," 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, 2013, pp. 478-492, doi: 10.1109/SP.2013.39.

[44] https://www.cs.huji.ac.il/project/Fairplay/FairplayMP/SFDL2.0.pdf, accessed (2020-12-08)