TECHNISCHE UNIVERSITÄT DARMSTADT

Bachelor Thesis

Generalizing Semi-Private Function Evaluation

Julian Götz Bieber

October 25, 2016

CRISP
Center for Research in Security and Privacy

Technische Universität Darmstadt
Center for Research in Security and Privacy
Engineering Cryptographic Protocols

Supervisors: Dr. Thomas Schneider
M.Sc. Ágnes Kiss

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Darmstadt, October 25, 2016

Julian Götz Bieber

Abstract

The cryptographic primitive Secure Function Evaluation (SFE) enables the evaluation of a function public function $f(x, y)$, under input parameters that are provided by different parties without them revealing their specific value. Restricting SFE further by requiring the function to be only known by one party is called Private Function Evaluation (PFE). Furthermore Semi-Private Function Evaluation (SPF-SFE) can help to increase the maximum function size which can be evaluated in a somewhat private manner. One way to achieve SPF-SFE is to split the function into blocks, as was shown in [PSS09]. However their implementation was crafted specifically for Yao's garbled Circuit protocol. Therefore there are protocols, namely Goldreich-Micali-Wigderson (GMW), by which it could not be evaluated.

We provide circuit designs for different Privately Programmable Blocks, which are constructed to work with the GMW protocol as well. Additionally we provide a toolchain to achieve SPF-SFE efficiently. Furthermore we implemented a tool used in the toolchain, which uses the aforementioned designs and can combine them with plain circuits and Universal Circuits (UCs) to build a single programmable circuit. A function designer can use the toolchain to balance out his needs for function privacy with performance.

Contents

1 Introduction

Secure Function Evaluation (SFE) is a cryptographic primitive that allows for a function $f(x, y)$ to be evaluated by two parties Alice and Bob. However there is the condition that only Alice knows x and only Bob knows y and neither is comfortable with revealing their value. One such situation in which SFE might be applied is that Alice and Bob want to meet each other in a given week. And since both are rather occupied they want to minimize travel time and because they are reasonably concerned about their privacy they want to avoid sharing their respective locations throughout the week. Therefore they need to evaluate a function which determines their optimal meeting place and time through an SFE protocol and their requirements will be met. Another, more theoretical example is the millionaire's problem as described in [Yao82a]. Here two millionaires want to compare their wealth without disclosing it to each other.

Private Function Evaluation (PFE) goes one step further and requires the function f to be private as well. However this condition also allows to simplify the function to be evaluated as $f(x)$ since the second input parameter can be hardcoded into the function itself. One example where PFE could be applied is a web based image manipulation service for privacy conscious users, that developed some proprietary filter. In this case the private function is the filter and the private input is the image.

However when the complexity of f increases the protocols that implement PFE get infeasible. Trading function privacy against evaluation performance is one way to diminish this effect and allow for more complex functions to be evaluated in a somewhat private manner. This concept is known as Semi-Private Function Evaluation (SPF-SFE), some aspects about the function do not necessarily have to be hidden, since an attacker would simply infer the information from the domain. For example in the web based image manipulation service mentioned above, an image filter is usually applied to every pixel in the image and takes the values of the surrounding pixels into consideration. A party trying to reconstruct the proprietary algorithm would most likely assume this fact even if the whole function is kept private. Therefore if the completely private function can not be evaluated, because of its size, and the less private version is small enough to be evaluated SPF-SFE is reasonable solution.

A very specialized version of SPF-SFE was used in [PKV+14]. They proposed a relational data base system which allows a user to issue queries whilst revealing the general structure of the query but keeping the values, tables and columns hidden. Furthermore it still allows the data base owner to restrict access to tables and columns on a user by user basis.

1.1 Contributions

In total there are three main contributions we made. The first of which is designing and implementing the pipeline of different tools which can be used to enable the easy design and efficient evaluation of semi private functions. We detail the design of this toolchain in Section 4.1. In this pipeline we use two existing tools, one to translate a function into a circuit, which we mention in Section 2.3. The second tool compiles a circuit definition into a Universal Circuit (UC) and we refer to it in Section 2.3.1.

Further we designed three Programmable Blocks which can be used in a semi private circuit. Namely one block which encapsulates the boolean operators AND, OR, XOR, $NAND$, NOR and $XNOR$. The second block we designed is one, which can either be used as a $<$, $>$, $=$, $\leq$ or a $\geq$ comparison. And the final block can either compute an addition or a subtraction. We describe each of them in detail in Section 4.2.

Our final contribution is a tool which translates a high level circuit design, containing Programmable Blocks, UCs and plain circuits into one circuit. We detail the implementation of the tool in Section 5.1.

1.2 Outline

In the following in Chapter 2, we provide the necessary preliminaries for later sections, such as a short definition for Boolean circuits in Section 2.1, a overview on SFE in Section 2.2, and its extension PFE in Section 2.3.

Furthermore in Chapter 3.1, we introduce SPF-SFE by highlighting the concept of a Privately Programmable Block in Section 3.1 and discussing a framework in which it was implemented in Section 3.2.

Then we present the design of our two contribution in Chapter 4, namely a pipeline for SPF-SFE in 4.1 and the circuits we used for the Privately Programmable Blocks in Section 4.2.

Next in Chapter 5 we discuss the implementation of our tool for Generalizing SPF-SFE in Section 5.1 and the testing it underwent in Section 5.2.

And in Chapter 6 we present one example for the use of our aforementioned tool in multiple configurations, namely using a circuit constructed purely from Privately Programmable Blocks in Section 6.1, a plain circuit in Section 6.2, a combination of Privately Programmable Blocks with a plain circuit in Section 6.3, a UC in Section 6.4 and a combination of Privately Programmable Blocks and one UC in Section 6.5.

Finally we conclude in Chapter 7.

2 Preliminaries

In this chapter we describe the basic concepts used in the design and implementation of our tool for generalising SPF-SFE. First we introduce Boolean Circuits in Section 2.1, secondly we describe the concept of SFE in Section 2.2. We mention two protocols that can be used for SFE, first of which is Yao's Garbled Circuit Protocol in Section 2.2.2 and second of which is the GMW Protocol in Section 2.2.3. In Section 2.3 we discuss PFE and one method to achieve it, namely UCs in Section 2.3.1.

2.1 Boolean Circuits

A given function $f : 0,1^n \rightarrow 0,1^m$ can be represented as a Boolean circuit with n input wires, m output wires and a number of gates k. Whereas a gate is a simple Boolean function with two inputs and one output. A gate can be defined through a truth table, listing the output for every combination of input values. For two input wires and one output wire the table has four entries. If for every truth table there is a combination of gates so that its evaluation result is equivalent to the truth table, the set of gates is called functional complete. The functional complete set of gates we will focus on is $\{XOR, AND\}$. Figure 2.1 depicts a XOR gate and Figure 2.2 represents an AND gate.

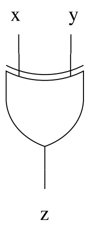


Figure 2.1: A single XOR gate

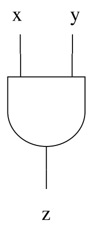


Figure 2.2: A single AND gate

2.2 Secure Function Evaluation

SFE is a cryptographic primitive which allows to solve the problems similar to the following situation. Two parties want to evaluate a function $f(x, y)$ on their private inputs, under the added condition that each party only has information about its own input and no trusted third party is to be used.

For example we take Alice and Bob, who want to arrange a meeting. Obviously both know their own schedule and would greatly prefer not to reveal their entire schedule. Therefore SFE can be used to calculate the set intersection between the timeslots in which they respectively are available. There exist interactive protocols, such as Yao's Garbled Circuit Protocol or the GMW protocol, that work on a Boolean circuit representation of the function which are used to solve SFE. We discuss Yao's Garbled Circuit Protocol in Section 2.2.2 and GMW in Section 2.2.3. One of the primitives used in the two protocols is Oblivious Transfer (OT), which we describe in Section 2.2.1.

2.2.1 Oblivious Transfer

In a 1-out-of-n OT protocol there are two parties, the sender and the receiver. The sender has n messages and the reciever knows the index of the message he should recieve. The goal however is for the messages not selected by the receiver to remain hidden. OT can be executed very efficiently due to the OT extensions from [IKNP03; ALSZ13].

2.2.2 Yao's Garbled Circuit Protocol

In Yao's garbled circuit protocol [Yao86; Yao82b] there are two parties involved. One is called the function provider and the other one is the function evaluator. The function provider knows the circuit representation of the function and his own input. The function evaluator only knows his input. For every gate in the circuit the function provider creates an encrypted truth table by assigning seemingly random values for the possible input and output values. Those encrypted truth tables and the encrypted input values from the function provider are transferred to the function evaluator, who uses 1-out-of-2 OT, which is described in Section 2.2.1, to get the encrypted values corresponding to its input. The function evaluator now evaluates the encrypted gates with the encrypted values. When the evaluation is finished the function provider reveals the plain value corresponding to the encrypted output.

Furthermore there are several optimizations for this protocol. For example in [KS08a] a method was proposed which allows to evaluate XOR gates without communication between the function evaluator and the function provider.

2.2.3 GMW Protocol

Another protocol for SFE is the GMW protocol [GMW87], which is based on secret sharing. In contrast to Yao's protocol it only allows for AND gates and XOR gates. Every input and output per gate is secret shared between the two parties. Therefore each party i has a seemingly random value v_i where $v = v_0 \oplus v_1$.

XOR gates can be evaluated as follows. The secret shared output z of the XOR gate has to fulfill $(z_0 \oplus z_1) = (x_0 \oplus x_1) \oplus (y_0 \oplus y_1)$. Which is equivalent to $(z_0 \oplus z_1) = (x_0 \oplus y_0) \oplus (x_1 \oplus y_1)$. Therefore both parties can evaluate the XOR gate locally without communication by calculating $z_i = x_i \oplus y_i$.

There are two methods for the evaluation of AND gates. The first of which is based on OT, the second of which utilizes so-called Multiplication Triples.

The OT based method works as follows. The inputs x and y are each secret shared as described above. Party 0 and party 1 use a 1-out-of-4-OT protocol, as described in Section 2.2.1, to calculate their respective secret shared part of the result. Where party 0 acts the the receiver in the OT protocol and party 1 acts as the sender in the OT protocol. Party 1 chooses a random value as its part of the result z_1 and provides four inputs to party 1 via OT. Whereby party 0 receives his part $z_0 = z_1 \oplus ((x_0 \oplus x_1) \oplus (y_0 \oplus y_1))$.

The Multiplication Triple based method works as follows. When 3 values a, b, and c satisfy $c = a \oplus b$ they are called a Multiplication Triple. When all those values are secret shared between both parties, which can be achieved via OT, they can be used to calculate the secret shares for the AND gate output. Both parties calculate $d_i = x_i \oplus a_1$ and $e_i = y_i \oplus b_i$ and exchange the results plainly, which enables them to calculate $d = d_0 \oplus d_1$ and $e = e_0 \oplus e_1$. With d and e calculated they can calculate their respective part of the result as follows:

$$z_0 = (d \oplus e) \oplus (b_0 \oplus d) \oplus (a_0 \oplus e) \oplus c_0 \tag{2.1}$$

$$z_1 = (b_1 \oplus d) \oplus (a_1 \oplus e) \oplus c_1. \tag{2.2}$$

2.3 Private Function Evaluation

In the example mentioned in Section 2.2 the function was allowed to be public knowledge. However not every situation warrants this condition. PFE allows to evaluate a secret function on a secret input. The specific methods for PFE have a major drawback, as they become impractical for more complex functions. In Section 2.3.1 we give an overview of one method for PFE, namely UCs.

2.3.1 Universal Circuits

The idea behind UCs is that given a specific circuit, a programmable circuit can be constructed that can evaluate any circuit of the same size, by assigning values to the programming bits. Since the UC does not leak information about the specific function that is to be evaluated it can be shared between both parties. The private evaluation of the function can then be seen as an application of a circuit based SFE protocol where the circuit is the UC and one parties' input is the programming bits that define circuit C.

One construction scheme was proposed in [KS08b] and is implemented in the Fairplay extension FairplayPF. For a circuit of size n it increases the circuit size in the order of $O(n * \log^2 n)$. A different and better scaling design was proposed in [KS16; Val76] and was implemented the framework detailed in [KS16]. In the circuit size it scales in the order of $O(n * \log n)$.

3 Semi-Private Function Evaluation

One approach to improve the scaling in the function size mentioned in Section 2.3.1 is to split the function into parts which absolutely have to be private and parts where secrecy is less of a concern or none at all. Those techniques are called SPF-SFE. It allows to make a fine-grained trade-off between the privacy of the function and the evaluation performance as shown in [PSS09].

If the use case of the function reveals some internals of the function and the performance is of importance there is little reason to hide that specific part. For example in a credit checking algorithm it is expected that the age of the person requesting the credit is compared to some constant. In Section 3.1 and 3.2 we describe the contributions from [PSS09].

3.1 Privately Programmable Blocks

The concept of a Privatlely Programmble Block (PPB) was proposed in [PSS09]. Generally a block is a boolean function and can be represented as a circuit. Therefore a PPB is a boolean function which depending on one of the inputs behaves equal to one function from a set of functions. The private part is achieved through the protocol by which the circuit is evaluated. For every PPB $b \in B$ where $B : \{0,1\}^{\log_2 |F|} \times \{0,1\}^n \rightarrow \{0,1\}^m$ there is a set F of functions and for every $f \in F$ where $F : \{0,1\}^n \rightarrow \{0,1\}^m$ there is a programming p so that $\forall x \in \{0,1\}^n : b(p,x) = f(x)$. For example F can be the set of comparisons like $\{<,>,=\}$.

The PPB has a privacy of $\log_2 |F|$ bits, since it can, depending on an input, evaluate equal to every function in F and the set of functions is revealed whereas the input is kept private.

A UC, as introduced in Section 2.3.1, can be viewed as the most generic version of such a PPB. Therefore it is only natural when constructing an entire function to be able to use UCs and PPBs alongside one another to hide what is required to be hidden and to reveal what is assumed to be known.

[PSS09] includes implementations for the following PPBs:

1. A comparison block which encompasses $\{<,>=,\leq,\geq,\neq\}$,
2. An addition and subtraction block, which is comprised of the respective functions,

3. A boolean block, which can evaluate equal to either AND, XOR, OR, $NAND$, NOR and $XNOR$,
4. A permutation block, which takes n input wires m of those input wires are forwarded as output wires, however the order of the outputs depends on an programming input parameter,
5. A selection block, takes n input wires and, dependent on a programming input paramter outputs one of the input wires,
6. A Y-multiplexer, which is a special case of the selection block, which takes exactly two input wires,
7. An X-switch, which is a special case of the permutation block, it has two input and two output wires and either inverses the order or keeps it as it is,
8. A UC, which was constructed according to the scheme from [KS08b] and therefore has a size of $O(n \log^2 n)$.

However next to the PPBs they implement multiplication as a non programmable block.

The PPBs are constructed as circuits. Therefore they can be evaluated with circuit based SFE algorithms, for example Yao's Garbled Circuit Protocol, as described in Section 2.2.2. However since the function is not entirely private using a SFE algorithm to evaluate a circuit containing PPBs leads to SPF-SFE.

3.2 FairplaySPF Framework

In [PSS09] they implement SPF-SFE based on the PPBs and UCs, according to the construction from [KS08b] in a framework called FairplaySPF. To describe the function and its partitioning into blocks they proposed a description language called SPBDL. Every line in SPBDL either specifies a set of input wires which together represent an input value, a multi-wire which they call vector and condenses multiple wire sets into a single wire set, a block as described in Section 3.1 or a set of output wires.

The file begins with an input specification, which consists of a number of lines, each establishing their index and their bitsize. After that each line defines a block and which blocks or inputs serve as its input. And the final lines each indicate a block index which is used as an output.

This SPBDL circuit description can be interpreted by FairplaySPF compiler to generate an SHDL circuit specification which in turn can be evaluated by the FairplaySPF runtime environment. FairplaySPF uses Yao's Garbled Circuit Protocol to evaluate a circuit on private inputs. Yao's Garbled Circuit Protocol allows arbitrary function tables for every gate, therefore the Programmable Block constructions from [PSS09] make use of those arbitrary function tables to minimize the size of the circuit. However the Goldreich-Micali-Wigderson (GMW) Protocol does not offer this freedom which means their constructions can not be used in GMW without further transformations.

Therefore we generalize the PPB construction as shown in Section 4.2 to only use gates which are supported by GMW, namely XOR and AND gates.

4 Design

In this chapter we first discuss the pipeline for SPF-SFE we designed in Section 4.1. And second we present the circuits for the Privately Programmable Blocks in Section 4.2.

4.1 Semi-Private Function Evaluation Pipeline

To evaluate a given function in a semi-private manner we use multiple different tools in conjunction. First there needs to be a tool, which translates a given function into a circuit representation. Second we use a tool to compile a plain circuit, such as the one provided by

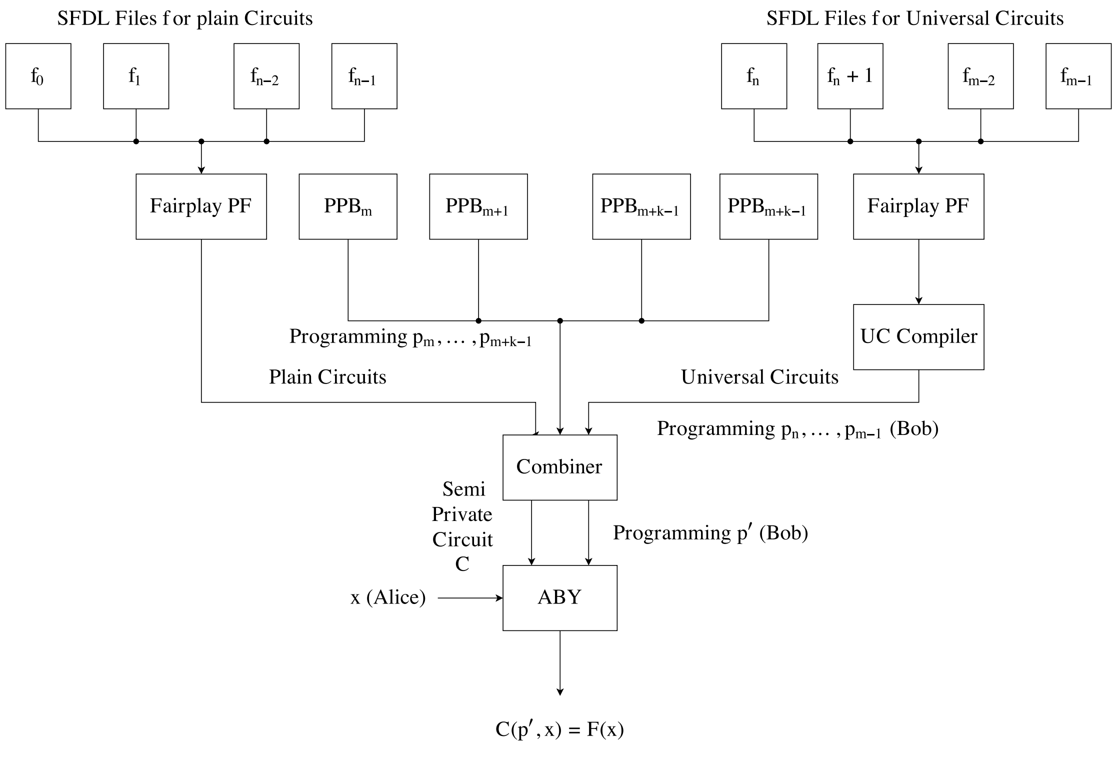


Figure 4.1: Semi-Private Function Evaluation Pipeline

the first tool, into a UC. And third we combine plain circuits, UCs and the Programmable Blocks that are defined in Section 3.1 and designed in Section 4.2.

The Fairplay framework was proposed in [MNPS04] and was improved in [**FairplayImprevement**], it includes tools to compile a function into a circuit description and to evaluate those circuits privately. It was extended in [KS08b] which, importantly for us changes the circuit compiler to only use gates with two input wires. Furthermore the extension allows to build UCs according to the construction from [KS16]. The extended framework is called FairplayPF, and we use its function to circuit compiler.

The circuit from the FairplayPF compiler then can either be translated into a UC with the tool that implements the state-of-the-art UC construction from [Val76; KS16], or it can directly be used in the combined circuit. The third tool in the chain is described in Section 5.1, it generates the generalized Privately Programmable Blocks and handles the combination of different blocks. It writes two separate files, one containing the circuit and one containing the corresponding programming.

The complete toolchain is depicted in Figure 4.1. There are m functions, all of them are compiled with the FairplayPF compiler. $n \leq m$ of the circuits are used as plain circuits, which is visualized on the left side of Figure 4.1 and the remaining $m - n$ circuits are further compiled by the UC Compiler from [KS08b], which is represented on the right side of Figure 4.1. Furthermore there are k Privately Programmable Blocks, depicted in the middle of Figure 4.1. The plain circuits, the Privately Programmable Blocks, the UCs and the programming for both the Privately Programmable Blocks and the UCs $p_n, \ldots, p_{m+k}$ are combined by hand using a high level block description, which is detailed in Section 5.1.1. The Combiner then takes this description as an input and produces one circuit C and the corresponding programming p' automatically.

In the end we use the ABY secure computation framework from [DSZ15] with state-of-the-art optimizations to, under Alice's input x, evaluate $C(p', x)$.

4.2 Generalized Privately Programmable Blocks

In this section we discuss the design of the three Programmable Blocks implemented in our tool. We describe the Boolean Operator block in Section 4.2.1, the Addition and Subtraction block in Section 4.2.2 and the Comparison block in Section 4.2.3. We require every circuit to only contain AND and XOR gates. For every block the goal is to first and foremost minimize the amount of AND gates in the circuit and to a lesser degree minimize the amount of XOR gates. The correctness of every block was thoroughly tested as described in Section 5.2.2.

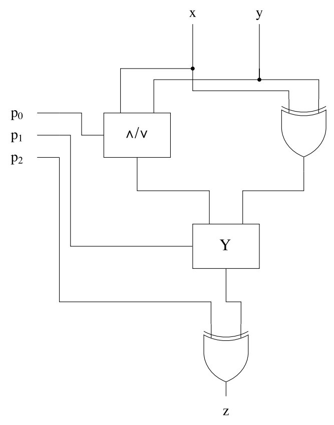


Figure 4.2: Boolean Operator Circuit

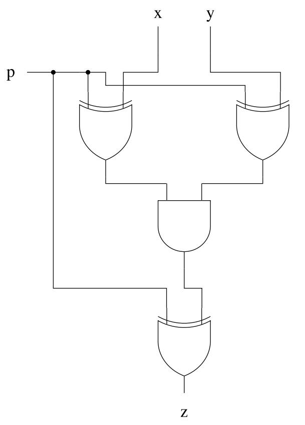


Figure 4.3: ∧/∨ Circuit

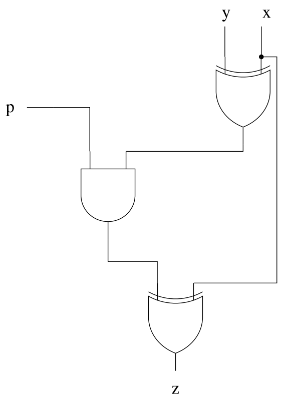


Figure 4.4: Y multiplexer Circuit

4.2.1 Boolean Operator

The Boolean Operator block evaluation depends on the three input bits and evaluates dependently either equivalent to an AND, OR, XOR, $NAND$, NOR or $XNOR$ gate. The general structure of the block is depicted in Figure 4.2.

The concept behind the ∧/∨ circuit, which is represented in Figure 4.3, is De Morgan's law $\neg(x \vee y) = \neg x \wedge \neg y$, which equals $x \vee y = \neg(\neg x \wedge \neg y)$. The negations are implemented as XOR gates, taking the input to be negated and $TRUE$ as inputs. However there is no constant input in the circuit, to make it programmable the programming bit is used instead of the constant $TRUE$ in the negation. Therefore this part switches between AND and OR depending on the input p_0.

$$\wedge/\vee(x, y, TRUE) = x \vee y, \tag{4.1}$$

$$\wedge/\vee(x, y, FALSE) = x \wedge y. \tag{4.2}$$

The Y-multiplexer follows the construction shown in [KS08b; KS08c] and is shown in Figure 4.4. If $p_1 = TRUE$ the result of the Y-multiplexer is the result from the XOR evaluation and if $p_1 = FALSE$ the result of the Y-multiplexer is the result from the AND/OR evaluation.

The last XOR represents a conditional negation of the result of the Y-multiplexer. If $p_2 = TRUE$ the former result is negated, otherwise it remains the same.

4.2.2 Addition/Subtraction

The programming bit p must switch the functionality of the Addition/Subtraction block between an addition and a subtraction. Given a number of input bits n, the number of output bits $m = n + 1$, to avoid over and underflow errors.

The block consists of n one bit addition circuits, one circuit to calculate the most significant result bit and a conditional negation of the entire second operand. The structure is depicted in Figure 4.5. Every one bit addition circuit has 3 input wires and 2 output wires. Each of the input wires represent either one bit of the first operand x_i, one conditionally negated bit of the second operand y_i and the carry output of the previous one bit addition circuit c_{i-1}, whereas $c_{-1} = p$. Per one bit addition circuit one of the output wires represent one output bit of the entire block z_i and the other one is the carry output c_i. The aforementioned conditional negation is a *XOR* gate which takes one bit of the second operand y_i and the programming bit p as inputs.

The most significant output bit, which serves as an over and underflow protection, depends on the last two carry bits c_{n-1} and c_{n-2} and the highest result bit z_{n-1}. It is calculated by $z_n = c_{n-1} \oplus c_{n-2} \oplus z_{n-1}$.

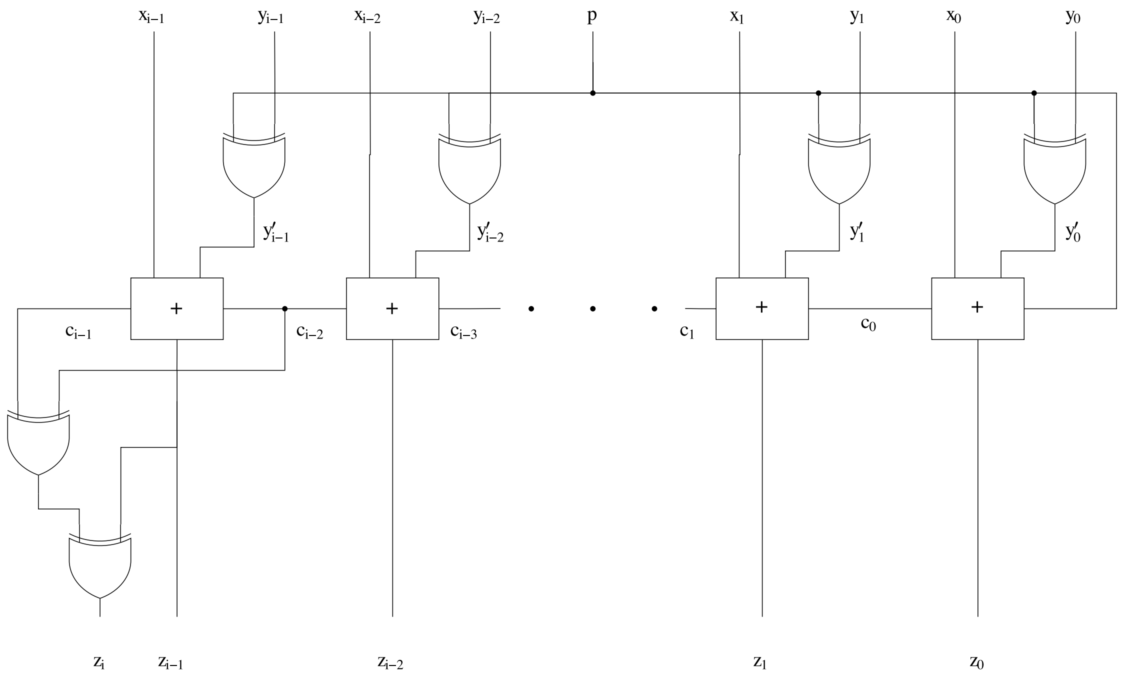


Figure 4.5: Addition or Subtraction Circuit

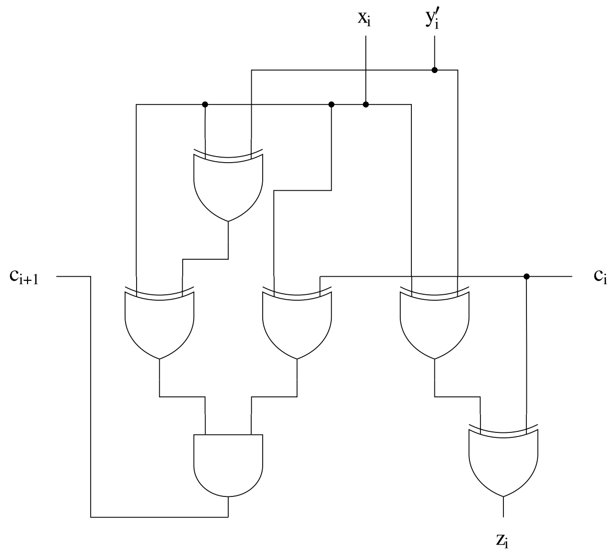


Figure 4.6: Addition Circuit

Every addition circuit calculates a 3 bit XOR, the result of which is is one bit of the result number. Additionally each addition circuit also calculates the next carry bit, the structure of the circuit is illustrated in Figure 4.6, which is taken from [BPP00].

The input numbers x and y are expected to be represented as 2-complements. Under this representation multiplying a number by -1 is realized as follows: $y * -1 = (\neg y) + 1$. Programming this block is conceptually realized by either calculating $z = x + y$ or $z = x + -1 * y$ which is equal to $z = x + (\neg y) + 1$. The bitwise not of the second operand y is realized with $y \oplus p$ in front of every addition and the plus one is realized by using p as the carry input of the first addition. Therefore if $p = TRUE$ the block subtracts y from x and if $p = FALSE$ it adds x to y.

4.2.3 Comparison

The Comparison block must, depending on the programming and given both inputs x and y, either calculate $x < y$, $x \leq y$, $x = y$, $x > y$ or $x \geq y$. The circuit used for the comparison is depicted in Figure 4.7. If the circuit in front of the final, on p_2 dependent, conditional negation is either the result of $x < y$, $x > y$ or $x = y$ all comparison operators are realized. From a high level perspective the block consists of one circuit which checks for equality, one circuit which calculates, depending on the first programming bit p_0, either $x > y$ or $x \leq y$ and logic to combine those results and is depicted in Figure 4.7. The result of the greater/less or equal circuit is combined with the negated result of the equal circuit by an AND, which leads to it calculating either $x < y$ or $x > y$. The Y-multiplexer chooses, depending on p_1 if

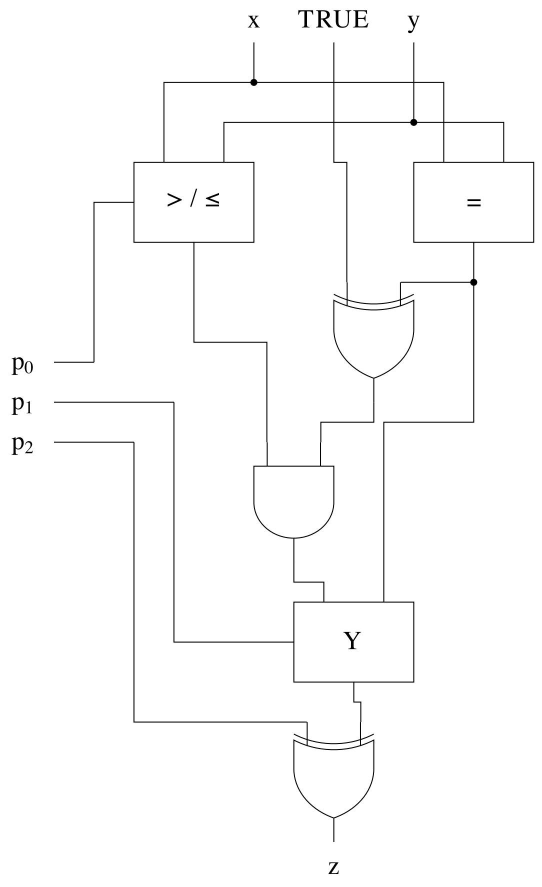


Figure 4.7: Comparison Circuit

either the result of the equal check or the less/greater check is used for the final conditional negation.

The sign of the result of $x - y$ shows if x is greater then y. Therefore if the most significant result bit is $TRUE$, x is less or equal to y otherwise x is greater y. To make that programmable between $x > y$ and $x \leq y$ the result is combined with p_0 by a XOR gate.

Two numbers are equal if all bits are equal. Therefore for all $0 <= i < n$ we calculate $(x_i \oplus y_i) \oplus TRUE$ and combine the results of the individual equal checks in a chain of AND gates which is depicted in Figure 4.8.

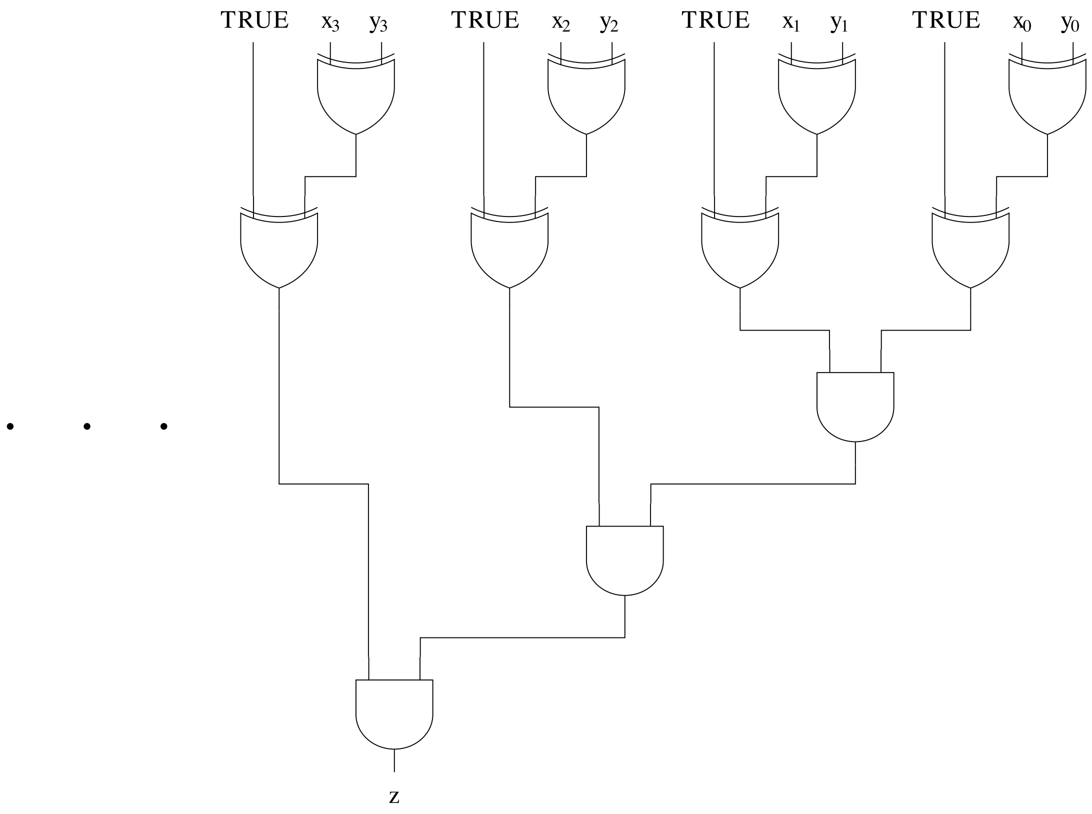


Figure 4.8: Equality Check Circuit

5 Implementation

In this chapter we provide details on the implementation of the tool for generalizing Semi-Private Function Evaluation and how both the implementation and the circuit design was implemented and tested. More specifically in Section 4.1 we describe about the tools we used to accomplish SPF-SFE and in Section 5.1 we provide an overview on the implementation of our Tool for generalizing Semi-Private Function Evaluation and the Programmable Blocks. Finally in Section 5.2 we describe how both, the tool and the circuits were tested for correctness.

For translating a function to a circuit we use the FairplayPF compiler, which was mentioned in Section 4.1. It creates a circuit which then can either be translated into a UC with the tool from [KS16], or it can directly be used in the combined circuit. The third tool in the chain is described in Section 5.1.

5.1 Tool for Generalizing Semi-Private Function Evaluation

In this section we first describe the input and output format of the tool in Section 5.1.1 and Section 5.1.2. Secondly in Section 5.1.3 we present how we convert a number of connected blocks into a circuit Furthermore we detail the general implementation of the programmable blocks in Section 5.1.4. Implementation details on the way UCs and plain circuits are handled are in Section 5.1.5 and Section 5.1.6 respectively.

5.1.1 Input Format

The definition of the combined circuit follows the SPBDL format, which was proposed in [PSS09] and which we slightly modified. The file format differs from the original in that we only UCs, Privately Programmable Blocks and plain circuits. It allows to specify a number of input values and their respective bit size, which defaults to 1. Following the input definition comes the high level circuit combination, which allows to specify blocks and the wiring between them. Whereas a block can either be a Programmable Block, as described in Section 4.2, their respective counterparts with a constant value for the second operand, a UC or a plain circuit. The output wires are then specified in the final lines.

The specification of a Programmable Block follow the pattern:

$$\begin{aligned} &< blockindex > \ [< blockidentifier >]\, out \ < outputbitlength > \ in\, [< firstoperandindex > \\ &\qquad < secondoperandindex >]\, p\, [< programmingidentifier >] \end{aligned} \tag{5.1}$$

For every Programmable Block there is a second version, which has the same circuit, but takes a constant as the second input parameter y. It follows the pattern:

$$\begin{aligned} &< blockindex > \ [< blockidentifier > c]\, out \ < outputbitlength > \ in \\ &\qquad [< operandindex >]\, p\, [< programmingidentifier > < constant >] \end{aligned} \tag{5.2}$$

UCs and plain circuits follow the same pattern, which is as follows:

$$\begin{aligned} &< blockindex > \ [< identifier >]\, out \ < outputbitlength > \ in\, [< operandindex >]\, p \\ &\qquad [< numberofgates > < pathtocircitfile >] \end{aligned} \tag{5.3}$$

We note, that the value for the number of gates does not affect the compilation as long as it does not contain whitespaces. This format limits UCs and plain circuits to only take one input parameter. To remedy that fact there is a vector block, which takes any number of inputs and concatenates them to a single output.

5.1.2 Output Format

When the tool for generalizing Semi-Private Function Evaluation is executed it produces two files as output. The first of which is the combined circuit and the second of which contains the programming for the used UCs and the used Privately Programmable Blocks. The first line of the circuit file contains a list of wire indices that represent the input values. After that there are a number of lines, marked with a P as the first character. Those lines represent the programming wires for the Programmable Blocks. Each of them contain only a wire index the corresponding value is in the programming file.

To differentiate between wire indices and values, values are a encoded as negative numbers, wheres $FALSE = -2$ and $TRUE = -3$. When all programming wires are defined, the next lines each represent a gate in the following pattern:

$$< gatetype >< firstinputwireindex >< secondwireindex >< outputwireindices > \tag{5.4}$$

Depending on the gate type there are either one or two output wire indices. AND, XOR, Y and U gates have one output wire and only X gates have two output wires. The third to last line represents the wire indices which are used as the output of the entire circuit. After that there are some statistics containing the number of AND and XOR gates used in the circuit. The non-XOR and non-AND gates each are an abbreviation for a combination of XOR and AND gates. A Y gate is implemented as two XOR and one AND gate, a U gate is implemented as six XOR and three AND gates and a X gate is implemented as three XOR and one AND gate as mentioned in [KS16].

The programming file contains all the programming values. There are two different line patterns used throughout the file. If the line contains only one number it represents a programming value for a UC. However if there are two numbers, the first represents the value and the second the index of the wire which the value is supposed to be used for.

5.1.3 Converting Blocks into a Circuit

Given the high level circuit specification, the `SPBDLParser` reads the lines one by one and constructs a list of `GenericBlocks` by identifying the type of the specified block. The parser returns the blocks and a list of indices, which specifies which blocks produce the output of the entire circuit.

The aformentioned `GenericBlock` is the base class of every block. It provides the interface which is used in the `CircuitBuilder` to combine the circuits for every block into a unified circuit. The two most noteworthy methods are `GenericBlock :: length` and `GenericBlock :: gatesWithOffset`. `GenericBlock :: length` returns the number of wires used in the sub circuit. And `GenericBlock :: gatesWithOffset` produces the list of gates which the subcircuit consists of. The returned list however depends on the offset and the input wire indices. However in case of the programmable blocks, the method `GenericBlock :: setProgrammingWires` must have been called before to ensure that the block knows which indices to use in place of the programming wires. The method adds the offset (the number of wires, already used in the circuit) to every new wire it introduces to the circuit and uses the input and programming wire indices at their respective positions in the subcircuit. It as well has the side effect, that it saves the output wire indices of the subcircuit.

The `CircuitBuilder` traverses the blocks three times. In the first traversal it gathers the input wire indices for the entire circuit. The second traversal adds the programming wires and their value to the circuit and saves the programming wire indices in the block. The third and final traversal retrieves the gates under the provided offset and the input wire indices and adds the length of the block to the current wire offset.

5.1.4 Programmable Blocks

There are three distinct sub classes of the `GenericBlock` which represent the three Programmable Blocks described in Section 4.2. Most of their methods are very simple in that they return either constants or values derived from the input bit length. Notably however are the specific implementations of `GenericBlock :: gatesWithOffset`.

The most forward implementation is the one for the `BooleanOperatorBlock`, the correspondig circuit of which was described in Section 4.2.1. It returns a hand written list of gates, that represent the exact circuit shown in Figure 4.2.

More interesting is the implementation for the `AddSubBlock`. It traverses the input wires from the least significant bit to the most significant bit and constructs the single bit adder circuits as shown in Figure 4.6. When all input wires are processed it adds the part of the circuit which calculates the most significant output bit.

The most complex of the three Programmable Blocks, the `ComparisonBlock` utilizes an `AddSubBlock`, which is programmed to be a subtraction to represent the $>\leq$ part of the circuit. After that the equality check circuit from Figure 4.8 and the result selection is appended to the list of gates.

5.1.5 Universal Circuits

The fourth distinct subclass of the `GenericBlock` is the `UCBlock`. When it is initialized the UC and the corresponding programming from the given path is parsed. We note that the file containing the programming has to have the same name as the circuit file except for the extension, which has to be '.programming'.

Both files are the output from the tool proposed in [KS16]. The circuit file contains first one line, which states the input wire indices, then a number of lines each describing one gate. The file ends with one line, defining the output wire indices. The `UCParser` has three duties. First it must parse the gates, where we consider input wires as gates. Second it must parse the programming file and third it must retrieve the output wire indices. When parsing a gate the first character indicates the type, the next two space separated numbers indicate the input wire indices and the following one or two, depending on the type, numbers indicate the output wire indices.

The list of `UCGates` returned by the static `UCParser :: parse` parser method is traversed in the `GenericBlock :: gatesWithOffset` implementation for the `UCBlock`. When processing a single `UCGate` it first retrieves the offsetted output indices of the two input wires for the current gate and uses them to build one `Gate` it also saves the offsetted output indices of the new gate.

5.1.6 Plain Circuits

The fifth and final distinct subclass of the `GenericBlock` is the `PlainCircuitBlock`. The idea behind the `PlainCircuitBlock` is similar to the aformentioned `UCBlock`. When it is initialized the circuit file is parsed and the `SHDLGates` are build. The `GenericBlock :: gatesWithOffset` implementation for the `PlainCircuitBlock` traverses the `SHDLGates` and transforms them to small circuits, which only contain AND and XOR gates. Those small circuits are defined in Table 5.1. The negations in those circuits are implemented as $x \oplus TRUE$ and the OR operators are implemented as described in Section 4.2.1.

Table 5.1: Transforming SHDL gates into small circuits.

SHDL identifer	Boolean function
0000	$x \oplus x$
0001	$x \wedge y$
0010	$x \wedge (\neg y)$
0011	$x \oplus FALSE$
0100	$(\neg x) \wedge y$
0101	$y \oplus FALSE$
0110	$x \oplus y$
0111	$\neg(\neg x \wedge \neg y)$
1000	$\neg x \wedge \neg y)$
1001	$\neg(x \oplus y)$
1010	$\neg y$
1011	$\neg(x \oplus y) \vee x$
1100	$\neg x$
1101	$\neg(x \oplus y) \vee y$
1110	$\neg(x \wedge y)$
1111	$\neg(x \oplus y)$

5.2 Testing

In this section we discuss how the tool described in Section 5.1 was tested. However we must first examine how a given circuit can be compared against a specified model, we do so in Section 5.2.1. Following in Section 5.2.2 we can delve into the the implementation behind the model for the three Programmable Blocks. Afterwards we describe the model generation for UCs in Section 5.2.3 and the model generation for plain circuits in Section 5.2.3. In the end we discuss how we combine those five models into a model for a combined circuit in Section 5.2.4.

5.2.1 Comparing Circuits against a Model

We utilize the case that two boolean functions $f : \{TRUE, FALSE\}^n \rightarrow \{TRUE, FALSE\}^m$ are equal if and only if for every combination of input values the result of both functions is equal. And because the number of combinations of input values is 2^n and therefore finite, we can simply evaluate both functions for every input and compare the outputs.

Therefore we must first evaluate the given circuit as a boolean function and second we must define a boolean function which behaves as a model. The specific models used are later discussed in Section 5.2.2, 5.2.3, 5.2.3 and Section 5.2.4. When parsing the circuit file we first extract five different lists, one for each gate type. Those gates contain their respective input and output indices and can be evaluated given two Boolean values. Afterwards we extract the wires from the circuit file. A wire represents a result from a gate evaluation which may not exist yet. The index within the list of wires corresponds to the wire index used in the circuit file.

Given the gates, wires and output indices the `Circuit` is constructed.

This `Circuit` can now be evaluated on a list of inputs and the programming from the programming file. The evaluation can be split into two distinct phases. First we try to evaluate every gate based on the current wires. If a gate was successfully evaluated the value of the destination wire is updated with the result. When all gates have been evaluated we check if all output wires contain a value. And repeat both steps until the condition is fulfilled.

5.2.2 Programmable Blocks

The simplest model of the three Programmable Blocks is the `BooleanOperatorBlock`, since both input operands already are present as Booleans it matches the programming and uses native implementation of the according boolean operator. The other two models, the one for the `AddSubBlock` and the one for the `ComparisonBlock` require a further step before they can call the matching native method to be evaluated. The lists of booleans which represent the input numbers have to be converted into integers. The conversion however is trivial, since the inputs are expected to be represented as two-complements.

5.2.3 Universal and Plain Circuits

There are two different concepts to be considered when we look at UC and plain circuit models. First the generation of a random UC or plain circuit and its evaluation. Generating random circuits is necessary because in contrary to Programmable Blocks there are too many possible UCs and plain circuits to check all of them. We note that we do not generate actual random UCs but rather a random circuit which is formatted the same way a UC would be formatted when it is constructed by the tool described in [KS16]. We justify that decision by noting that the unit under test does not have any understanding of the meaning of the circuit which it processes. Therefore if it can handle arbitrary circuits in the given format it should be able to handle arbitrary UCs as well.

To generate random circuit under a given number of input and output values, we choose a random number of gates greater or equal one and randomly choose which of the three gate types to use. The two input wire indices are randomly chosen between zero and the number of wires already used. The output wire index is simply the the current number of gates plus one, and potentially in case of X gates, the second output wire is the current number of gates plus two.

Evaluating the circuit works comparable to the evaluation of a circuit as described in Section 5.2.1.

5.2.4 Combined Circuits

When we have models to represent each of the three Programmable Blocks, UCs and plain circuits, we can combine them arbitrarily to create a multitude of different circuits. To generate an arbitrary circuit we first randomly choose the number of input and output values and the number of blocks used in the circuit. Based on these numbers we first randomly choose a bitlength for each input value and build as many random blocks as specified. To generate a random block we first choose the number of input and output bits. The relation between the input and output bitlength limits which block type may be chosen. UC blocks and plain circuit blocks are allowed for every case, if the number of input bits is one less then the number of output bits the Addition/Subtraction Block becomes an additional valid choice. If there is only one output bit, we include the Comparison Block in the pool of valid blocks. The Boolean Operator Block is taken into consideration if the number of input and output bits both equal one. To achieve a more equal distribution of block types we include a bias in the input and output size generation to specifically generate combinations that match the relation mentioned.

When the blocks are generated we need to wire the blocks between one another. For every block the number of input values and the size per value is known, therefore for every required value we take the blocks and circuit inputs with a lower index then the current block and shuffle them. Then we go over the previous blocks and input values and check whether or not the value may be chosen as an input for the current block. If the output size of the potential input block is less then the remaining input size of the current block we add it to the input. We repeat this process until the remaining input size is zero. We note that the first circuit input always has one bit, therefore this process is guaranteed to terminate.

If an input for a block consists of the outputs of multiple other blocks or circuit inputs they are aggregated as a vector block. As a final step we randomly choose blocks as circuit outputs.

When the circuit is generated we transform it into a string representation matching the format described in Section 5.1.1. This file now is compiled by our tool and produces a circuit and the corresponding programming. Finally the circuit in conjunction with the programming are checked against the model.

6 Credit Checking Results

As an example function we used a simple credit checking algorithm. Alice applies for a loan and wants to remain private. Bob has to check if Alice matches a set of criteria and wants to avoid those criteria becoming public knowledge. The function Bob uses decides, based on the age, the gender and the amount requested, whether or not to grant the credit. The age is represented as a seven bit integer, the gender is represented as a single bit and the requested amount is a 16 bit integer. The conditions for the allowance are as follows. The applying party must be over the age of 18 and if the person is female she has to be younger then 65. Furthermore the requested amount has to lie between zero and 50 currency units and the sum of the age and the requested amount has to be less then or equal to 85. If and only if any of the conditions is violated, the request will not be granted.

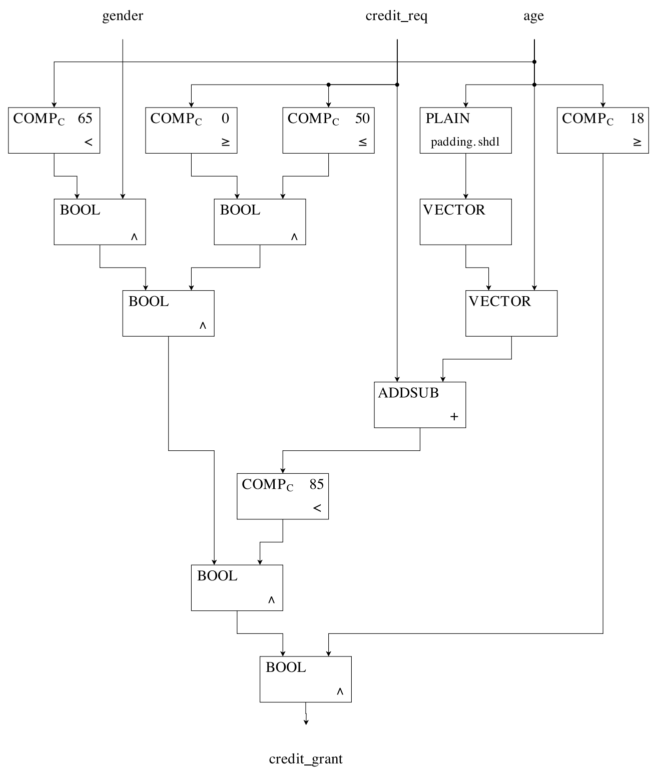
We created multiple different versions of the algorithm described above. The first version only uses Privately Programmable Blocks, as described in Section 4.2. We present the corresponding results in Section 6.1. As a lower boundary for the function privacy we simply use a plain circuit and we discuss the results in Section 6.2. The third version is a combination of a plain circuits and a number of Programmable Blocks, which we detail in Section 6.3. For the upper boundary for the function privacy we use a UC and the respective results are detailed in Section 6.4. As with the plain circuit version we built a circuit, that uses a UC in combination with multiple Programmable Blocks. It is described in Section 6.5. In the end we compare the results for the different circuits in Section 6.6.

6.1 Programmable Blocks

The circuit utilizes a Comparison Block to compare the age against the constant 18. A second Comparison Block compares the age against 65. Both results are combined by a Boolean Operator Block. Furthermore two Comparison Blocks compare the requested amount against zero and 50 respectively. The comparison results are again combined by a Boolean Operator Block.

An Addition/Subtraction Block is used to add the age to the requested amount. This block requires both input values to have the same number of bits. Therefore we have to pad the age depending on its sign. This we achieve by using a plain circuit, which consists of one XOR gate that combines the most significant bit of the age with $FALSE$. The result of the aforementioned Addition/Subtraction Block is compared, by a Comparison Block against the constant 85. Finally another Boolean Operator Block combines the previous results.

This circuit combination is depicted in Figure 6.1. In total there are 157 *AND* gates and 752 *XOR* gates used in the circuit and it provides 25 bits of privacy, there are 2^{25} functions which the circuit can represent.

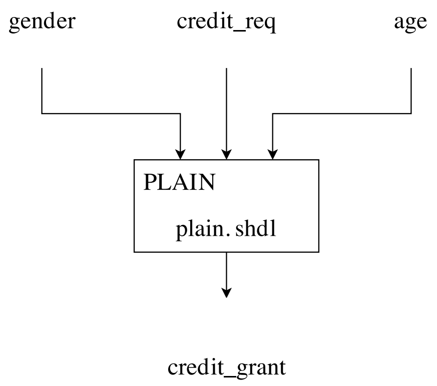


```
input alice [7] // age
1 input alice // 2 input alice [16]
// credit_req
3 block [compc] out 1 in [0] p [GE
18]
4 block [compc] out 1 in [0] p [L
65]
5 block [bool] out 1 in [4 1] p
[AND]
6 block [compc] out 1 in [2] p [GE
0]
7 block [compc] out 1 in [2] p [LE
50]
8 block [bool] out 1 in [6 7] p
[AND]
9 block [bool] out 1 in [8 5] p
[AND]
10 block [plain] out 1 in [0] p [0
padding.shdl]
11 vector [10 10 10 10 10 10 10 10
10]
12 vector [11 0]
13 block [addsub] out 17 in [2 12]
p [ADD]
14 block [compc] out 1 in [13] p [L
85]
15 block [bool] out 1 in [14 9] p
[AND]
16 block [bool] out 1 in [15 3] p
[AND]
17 output bob 16
```

Figure 6.1: Credit checking function, which uses only Privately Programmable Blocks

6.2 Plain Circuit

The plain circuit version of the function was compiled with the FairplayPF tool from [KS08b]. It executes the algorithm described in Section 6. The circuit combination is displayed in Figure 6.2. In total there are 133 *AND* gates and 251 *XOR* gates used in the circuit and it provides 0 bits of privacy, since whole circuit is revealed and it has no programming input.



```
0 input alice [7] // age
1 input alice
2 input alice [16] // credit_req
3 vector [1 2 0]
4 block [plain] out 1 in [3] p [0
plain.shdl]
5 output bob 4
```

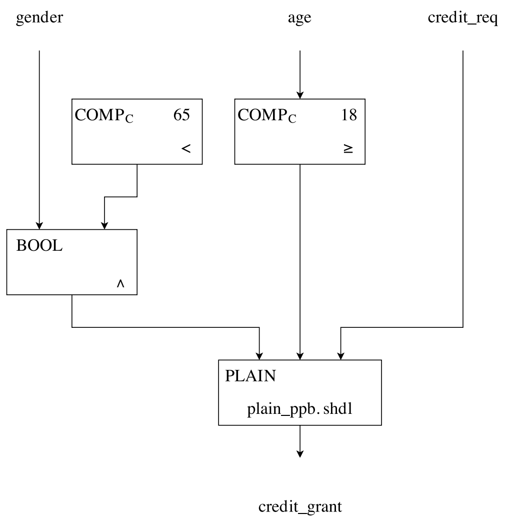
Figure 6.2: Credit checking function, which uses only a plain circuit

6.3 Plain Circuit with Programmable Blocks

In the version of the function, that combines a plain circuit with Programmable Blocks, we use two Comparison Blocks, one to compare against 18 and one to compare against 65. Furthermore we use a Boolean Operator Block to combine the comparison results. The plain circuit part checks if the requested amount satisfies the requirements stated in Section 6, the structure is detailed in Figure 6.3. In total there are 154 *AND* gates and 374 *XOR* gates in the circuit and because of the Privatley Programmable Blocks used it has 8 bits of privacy.

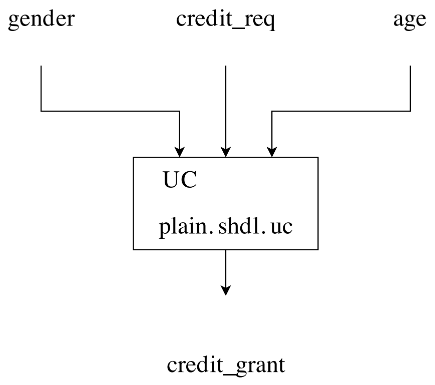
6.4 Universal Circuit

The UC for this version used the same SFDL file as a function description that we use in Section 6.2, we translate it to a circuit with the FairplayPF tool. The SHDL file, resulting from the translation was then compiled with the UC compiler from [KS16]. The combination is depicted in Figure 6.4. In total there are 8923 *AND* gates and 25456 *XOR* gates in the circuit and it provides 9219 bits of privacy.



```
0 input alice [7] // age
1 input alice
2 input alice [16] // credit_req
3 block [compc] out 1 in [0] p [GE
18]
4 block [compc] out 1 in [0] p [L
65]
5 block [bool] out 1 in [4 1] p
[AND]
6 vector [5 3 2 0]
7 block [plain] out 1 in [6] p [0
plain.shdl.uc]
8 output bob 7
```

Figure 6.3: Credit checking function, which uses Privately Programmable Blocks in conjunction with a plain circuit

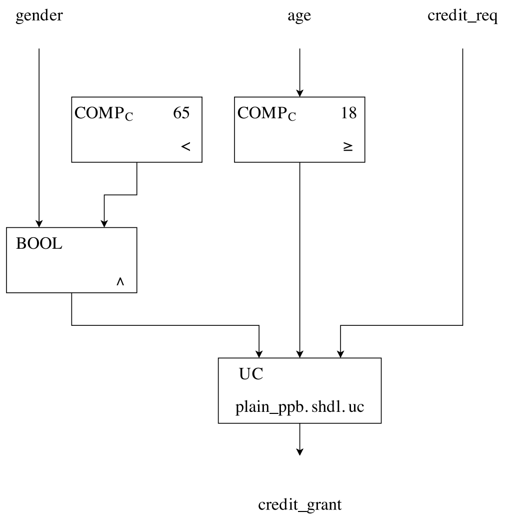


```
0 input alice [7] // age
1 input alice
2 input alice [16] // credit_req
3 vector [1 2 0]
4 block [uc] out 1 in [3] p [0
plain.shdl.uc]
5 output bob 4
```

Figure 6.4: Credit checking function, which uses only a UC

6.5 Universal Circuit with Programmable Blocks

The structure we use for this version of the circuit matches the one that uses plain circuits alongside Programmable Blocks, which is represented in Figure 6.5. The age limits are checked by two Comparison Blocks and one Boolean Operator Block. And the checks on the requested amount are executed by the UC, therefore Bob hides these checks completely. In total there are 7943 AND gates and 22667 XOR gates in the circuit and it provides 8183 bits of privacy.



```
0 input alice [7] // age
1 input alice
2 input alice [16] // credit_req
3 block [compc] out 1 in [0] p [GE
18]
4 block [compc] out 1 in [0] p [L
65]
5 block [bool] out 1 in [4 1] p
[AND]
6 vector [5 3 2 0]
7 block [uc] out 1 in [6] p [0
plain.shdl.uc]
8 output bob 7
```

Figure 6.5: Credit checking function, which uses Privately Programmable Blocks combined with a UC

6.6 Comparison of Results

The results of the different versions of the credit checking function are visualized in Figure 6.6. The version which only uses a UC is, as expected, the most privacy-preserving version. Whereas the least private version only uses a plain circuit. Between both extremes are the versions, which incorporate Programmable Blocks in the following order from least private to most private: a circuit which combines Programmable Blocks with plain circuits, one which only consists of Programmable Blocks and one where Programmable Blocks are linked with a UC.

To assess the performance, we look at the number of AND gates in the circuit. The performance ranking is the exact opposite of the privacy ranking. The best performance is achieved with the plain circuit, the worst performance is observed with the UC. Between them the ranking is as follows: a circuit combining an UC with Programmable Blocks, one which only uses Programmable Blocks and a circuit which incorporate a plain circuit among Programmable Blocks.

The better the privacy becomes the worse the performance, since the number of AND gates has the most significant impact on the performance.

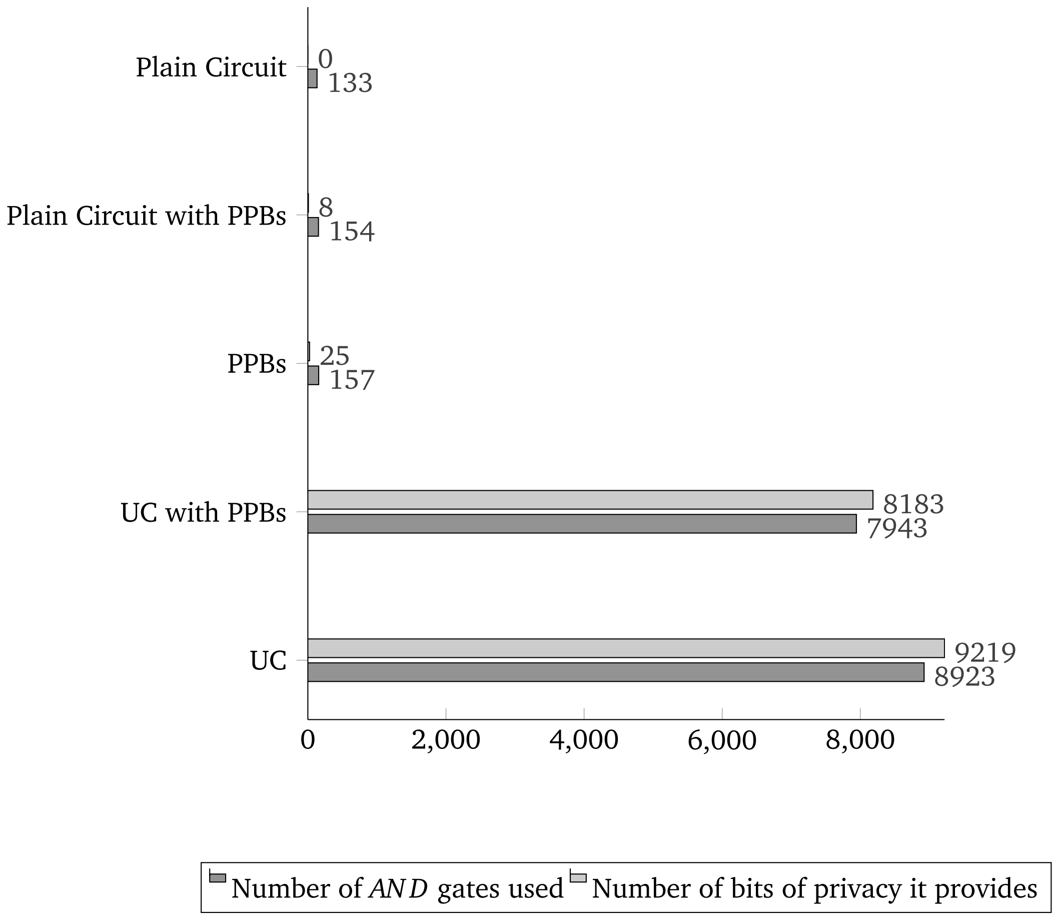


Figure 6.6: Results for the different configurations

7 Conclusion

We provide a toolchain to evaluate a function in a semi private manner, which we describe in Section 4.1. Furthermore we implemented one of the tools used in the toolchain, which is detailed in Chapter 5. Additionally we designed three generalized Privately Programmable Blocks, which are defined in Section 4.2, in a way so they can be evaluated by both the GMW protocol and Yao's garbled Circuit protocol.

The albeit small example, described in Chapter 6 shows the expected results, the less privacy-preserving the combination is the smaller the circuit becomes. Therefore using Privately Programmable Blocks amongst UCs and plain circuits appears to allow the circuit designer to balance function privacy with performance. However more and importantly bigger examples need to be tested to make a statement on the ratio of lost privacy versus gained performance is by using the concepts described in Chapter 4 and the implementation from Chapter 5.

Outstanding improvements for the tool specifically would be to include more different block types, for example arithmetic operations on floating point numbers or simply multiplications and divisions on integers and floating point numbers. The pipeline and the tool in conjunction can be improved by allowing both parties to specify parts of the function which is to be evaluated between them.

Furthermore the fact that the combination of Privately Programmable Blocks, plain circuits and UCs has to be optimized manually is more cumbersome than it should be. A method was proposed recently in [KKW16], which under the input of multiple circuits produces a single circuit which, dependent on a programming, can evaluate equal to each of the input circuits.

List of Figures

List of Tables

Abbreviations

SFE	Secure Function Evaluation
PFE	Private Function Evaluation
UC	Universal Circuit
SPF-SFE	Semi-Private Function Evaluation
PPB	Privatlely Programmble Block
OT	Oblivious Transfer
GMW	Goldreich-Micali-Wigderson

Bibliography

[ALSZ13] G. Asharov, Y. Lindell, T. Schneider, M. Zohner. “More efficient oblivious transfer and extensions for faster secure computation”. In: (2013), pp. 535–548 (cit. on p. 4).

[BPP00] J. Boyar, R. Peralta, D. Pochuev. “On the multiplicative complexity of Boolean functions over the basis”. In: *Theoretical Computer Science* 235.1 (2000), pp. 43–57 (cit. on p. 15).

[DSZ15] D. Demmler, T. Schneider, M. Zohner. “ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation”. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. 2015 (cit. on p. 12).

[GMW87] O. Goldreich, S. Micali, A. Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. 1987, pp. 218–229 (cit. on p. 4).

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, E. Petrank. “Extending Oblivious Transfers Efficiently”. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. 2003, pp. 145–161 (cit. on p. 4).

[KKW16] W. S. Kennedy, V. Kolesnikov, G. T. Wilfong. “Overlaying Circuit Clauses for Secure Computation”. In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 685 (cit. on p. 33).

[KS08a] V. Kolesnikov, T. Schneider. “Improved Garbled Circuit: Free XOR Gates and Applications”. In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*. 2008, pp. 486–498 (cit. on p. 4).

[KS08b] V. Kolesnikov, T. Schneider. “A Practical Universal Circuit Construction and Secure Evaluation of Private Functions”. In: *Financial Cryptography and Data Security, 12th International Conference, FC 2008, Cozumel, Mexico, January 28-31, 2008, Revised Selected Papers*. 2008, pp. 83–97 (cit. on pp. 6, 8, 12 sq., 29).

[KS08c] V. KOLESNIKOV, T. SCHNEIDER. “Improved Garbled Circuit: Free XOR Gates and Applications”. In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*. 2008, pp. 486–498 (cit. on p. 13).

[KS16] Á. KISS, T. SCHNEIDER. “Valiant’s Universal Circuit is Practical”. In: *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*. 2016, pp. 699–728 (cit. on pp. 6, 12, 19 sq., 22, 24, 29).

[MNPS04] D. MALKHI, N. NISAN, B. PINKAS, Y. SELLA. “Fairplay - Secure Two-Party Computation System”. In: *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*. 2004, pp. 287–302 (cit. on p. 12).

[PKV+14] V. PAPPAS, F. KRELL, B. VO, V. KOLESNIKOV, T. MALKIN, S. G. CHOI, W. GEORGE, S. BELLOVIN, A. KEROMYTIS. “Blind seer: A scalable private DBMS”. In: *IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE, 2014 (cit. on p. 1).

[PSS09] A. PAUS, A. SADEGHI, T. SCHNEIDER. “Practical Secure Evaluation of Semi-Private Functions”. In: *Applied Cryptography and Network Security (ACNS 2009)*. Vol. 5536. LNCS, 2009, pp. 89–109 (cit. on pp. e, 7 sq., 19).

[Val76] L. G. VALIANT. “Universal Circuits (Preliminary Report)”. In: *Proceedings of the 8th Annual ACM Symposium on Theory of Computing, May 3-5, 1976, Hershey, Pennsylvania, USA*. 1976, pp. 196–203 (cit. on pp. 6, 12).

[Yao82a] A. C. YAO. “Protocols for Secure Computations (Extended Abstract)”. In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. 1982, pp. 160–164 (cit. on p. 1).

[Yao82b] A. C. YAO. “Protocols for Secure Computations (Extended Abstract)”. In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. 1982, pp. 160–164 (cit. on p. 4).

[Yao86] A. C. YAO. “How to Generate and Exchange Secrets (Extended Abstract)”. In: *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. 1986, pp. 162–167 (cit. on p. 4).