TECHNISCHE
UNIVERSITÄT
DARMSTADT

Bachelor Thesis

Valiant's Universal Circuit - Towards a Modular Construction and Implementation

Daniel Günther

March 30, 2017

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, March 30, 2017

Daniel Günther

Abstract

Secure Function Evaluation (SFE) allows two parties to evaluate a function without sharing their inputs with one another, while Private Function Evaluation (PFE) hides additionally the computed function which one party provides, i.e. PFE is SFE with private functions. PFE can be efficiently reduced to SFE by using a Universal Circuit (UC) as public function for the computation.

A UC is a special Boolean Circuit, which is able to simulate every function of a certain size n. The first implementation of a UC was provided by Kolesnikov and Schneider (FC'08). Valiant (STOC'76) proposed the first and more efficient UC construction of size $\mathcal{O}(n \log n)$. He detailed two variants for his construction, one with 2-way and one with 4-way recursive structure. One of these two constructions - the 2-way split UC construction - was implemented by Kiss and Schneider (Eurocrypt'16). Lipmaa et al. (Eprint 2016/017) generalized this construction to a k-way split construction and concludes that the best performing construction would be the 4-way split construction. However, the 4-way split UC construction, achieving better concrete performance, was not implemented so far.

In this work, we design and implement a module for Valiant's 4-way split UC construction. This module can be embedded into a toolchain for PFE, which was provided by Kiss and Schneider (Eurocrypt'16) - designed for their own UC implementation. Additionally, we propose a modular consideration of UCs using the k-way split construction by Lipmaa et al. (Eprint 2016/017).

Contents

1 Introduction

Boolean Circuits are a common way to implement functions f, e.g. in a hardware embedding or for logical computing. While boolean circuits can handle variable inputs x to compute $f(x)$, a *Universal Circuit (UC)* is a special boolean circuit that can also handle variable functions f limited to a certain size. This means that a UC takes, additional to the function's input x, a representation of a function p_f as input and computes $UC(p_f, x) = f(x)$ like the implementation of a boolean circuit for function f also does.

The idea of UCs and the first constructions are provided by Valiant in [Val76]. Valiant designs two UC constructions - today known as 2-way and 4-way split constructions. Both constructions are asymptotically size-optimal [Weg87], i.e. the number of gates needed for the implementation of the UC is $\Omega(n \log n)$, where n denotes the number of inputs, outputs and gates of the computed function f. Recently, two papers in parallel, [KS16] and [LMS16], provide more detailed descriptions of the underlying algorithm and implementations of the 2-way split construction. The size for the 2-way split UC construction is larger than that of the 4-way split UC construction. However, there exists no implementation of the 4-way split UC construction so far.

A popular application for UC is *Private Function Evaluation (PFE)*, which is based on *Secure Function Evaluation (SFE)*. SFE allows two parties Alice and Bob to evaluate a public known function $f(x, y)$ without a third party. Thereby, they do not get to know each other's input, i.e. Alice only knows x and Bob only knows y but both learn $f(x, y)$ after the computation. [Yao82] proposes the millionaire's problem as application example for SFE. Two millionaires want to compare with a function $f(x, y)$ their fortune without providing their information to the other party. SFE of boolean circuits can be achieved using Yao's garbled circuit protocol [Yao82; Yao86] or the GMW protocol [GMW87].

However, there exist applications where function f shall be kept secret to the other party - called *Private Function Evaluation* (PFE). Here, only Alice knows the function $f(x)$ and Bob the input x but both want to compute $f(x)$. Only the result $f(x)$ is interpretable for Alice without knowing Bob's input x. PFE can be achieved by using SFE with a UC as public function. The inputs of the UC are the representation of the private function p_f as well as the secret input x. Therefore, we need well defined representations p_f of functions f and an efficient construction of a UC such that $UC(p_f, x) = f(x)$ can be computed. An application for PFE is the credit checking scheme provided by Frikken et al. [FAZ05]. In this application, Bob is the bank and Alice applies for a credit. Bob owns a secret function that checks if Alice is eligible to receive the credit from Bob. This function needs Alice's secret financial information. Using PFE allows both parties to keep their information secret and Bob knows more about Alice's financial status. There exist specific protocols for PFE that achieve better performance than UCs [MS13; MSS14], for further details, the reader is referred to [KS16].

Besides PFE, there are various applications for UCs. We provide some of them in the following, and refer to [KS16; LMS16] for further details on applications.

[GGPR13] provide their *Quadradic Span Programs* as a new characterization of the NP class. They use a UC to reduce the verifier's preprocessing step by setting up the *common reference string*.

[PKV+14; FVK+15] use a UC for hiding queries in a *Database Management System (DBMS)*. However, there exist queries, which do not hide the circuit topology. Therefore, [PKV+14] improve the Blind Seer DBMS by using a modified simpler UC for evaluating those queries.

1.1 Contributions

We design and implement Valiant's 4-way split UC construction which can be embedded into the toolchain of [KS16]. We use the approaches of [LMS16] to design our modular UC construction in Section 4.1. The implementation details are provided in Chapter 5.

Additionally, we improve the work of [LMS16] and provide an approach for implementing a k-way split UC construction by splitting the most difficult construction task into two subtasks which is detailed in Section 4.2.

1.2 Outline

In Chapter 2, we provide our notations and basic concepts, containing boolean circuits in Section 2.1, the necessary graph theory about directed acyclic graphs in Section 2.2, methods for abstracting boolean circuits and graphs using blocks in Section 2.3 as well as methods for SFE in Section 2.4.

Chapter 3 contains the related work for UCs. We firstly explain basic concepts for UCs in Section 3.1. Thereafter, we detail Valiant's UC construction [Val76] step by step in Section 3.2. This also contains the generalized k-way split construction [LMS16]. The last section of this chapter (Section 3.3) summarises the UC construction of [KS08a] and shortly introduces the hybrid UC constructions provided by [KS16] in Section 3.3.1.

The design of our 4-way split construction is given in Chapter 4, especially our gained modularity of Valiant's UC construction [Val76] in Section 4.1 and an approach for the edge-embedding task in Section 4.2.

In Chapter 5, we explain some details of our implementation, specifically, the toolchain for UCs in a PFE setting of [KS16] in Section 5.1 as well as our realization of the UC module which can embedded into their toolchain in Section 5.2.

Finally, we present our results in Chapter 6. Therefore, we provide a formula which calculates the exact size of our implementation and for any k-way split UC construction in Section 6.1. Additionally, we compare the number of AND gates between the UC construction of [KS16] and that of ours in Section 6.2. Furthermore, we give directions for future work in Section 6.3.

2 Preliminaries

In this chapter we introduce basic concepts and notations as well as common definitions used later for constructing a UC. We explain all concepts that are necessary for the understanding of the topic.

2.1 Gates and Circuits

In case of SFE, a common way to define functions are *Boolean Circuits*. A boolean circuit $C_{u,v}^{k,n}$ has u inputs, v outputs and k distinguished *gates*, denoted by $G_i^{n_i \leq n}$ for $i < k$. Such a gate G^n is the implementation of a boolean function $g^n : \{0,1\}^n \rightarrow \{0,1\}$. We denote by C_f the boolean circuit $C_{u,v}^{k,n}$ that computes the function $f : \{0,1\}^u \rightarrow \{0,1\}^v$. Every boolean circuit $C_{u,v}^{k,n}$ can be reduced to a boolean circuit $C_{u,v}^{k,2}$ (Note: we denote this special case with $C_{u,v}^{k}$ in the following). Therefore, we have to replace every gate $G_i^{n_i > 2}$ by multiple gates G_j^2. This can be done using Shannon's expansion theorem [Sha49] as shown in Equation (2.1).

$$f(x_1, \ldots, x_n) = (\overline{x_1} \wedge f(0, x_2, \ldots, x_n) \vee (x_1 \wedge f(1, x_2, \ldots, x_n))) \tag{2.1}$$

More details about this process can be found in [Sch08].

These modified boolean circuits will be used in Chapts. 3 and 4.

Because of the properties of the constructions described later and due to using $C_{u,v}^{k}$ boolean circuits we only need to consider gates G^2 with 2 inputs. If C has a gate G^1 with one input, we can extend it to a G^2 gate by adding a dummy input to G^1. A gate G^2 can be easily defined with its so-called gate table. We give some examples in Tab. 2.1.

2.2 Graph Theory

This section gives the preliminaries of Valiant's UC construction [Val76]. At the end of this section we show an example.

The UC construction of Valiant [Val76] is based on graph theory. We denote by $G = (V, E)$ a *directed graph* with a set of *nodes* V and *edges* $E \subseteq V \times V$. A sequence $(a_1, \ldots, a_n)$ is called a

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

(a) XOR gate

x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1

(b) AND gate

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

(c) OR gate

x_1	x_2	x_1 XNOR x_2
0	0	1
0	1	0
1	0	0
1	1	1

(d) XNOR gate

Table 2.1: (a) - (d) show implementation tables of the most common G^2 gates.

path if $\forall i \in \{1,\dots,n-1\} : (a_i, a_{i+1}) \in E$. In case of $a_1 = a_n$, we call the path $(a_1,\dots,a_n)$ a *cycle*. A graph is *acyclic* if it has no cycles.

In general we can characterize a node by its incoming and outgoing edges. The number of incoming [outgoing] edges is called *indegree* [*outdegree*]. A graph has *fanin* [*fanout*] p if the indegree [outdegree] of all its nodes is at most p. In the following, we denote by $\Gamma_p(n)$ the set of all acyclic graphs with fanin and fanout p having n nodes.

Every graph of fanout $p > 2$ can be reduced to a graph with fanout 2 by adding one node for each additional edge. [KS16; Val76] describe this process in detail.

Let $G = (V,E) \in \Gamma_p(n)$. A function $\eta : V \to \mathbb{N}$ is called a *labelling*. We want to define an order to η such that $(a_i, a_j) \in E \Rightarrow \eta(a_i) > \eta(a_j)$ and $\forall a_1, a_2 \in V : \eta(a_1) = \eta(a_2) \Rightarrow a_1 = a_2$. The value range of η is $\{1,\dots,n\}$. This order is called *topological order* and can be found with computational complexity $\mathcal{O}(n + pn)$.

In Section 2.1 we define a boolean circuit $C^k_{u,v}$ with u inputs, v outputs and k gates (we note that each gate has at most 2 inputs). We can transform a $C^k_{u,v}$ into a $\Gamma_2(u+v+k)$ graph by creating a node for each input, output and gate. Creating an edge for each wire in $C^k_{u,v}$ finishes the transformation.

2.3 Blocks

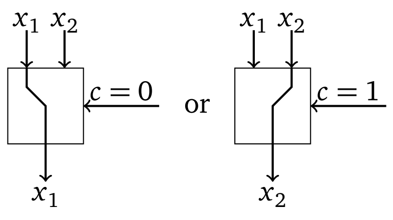
The overview of big circuits and graphs with many nodes can become very unclear. In this section, we provide some methods to make circuits and graphs more modular.

A *block* $B_{u,v}(f)$ with u inputs $(x_1, \dots, x_u)$ and v outputs $(y_1, \dots, y_v)$ is the implementation of a boolean circuit $C_f = C^k_{u,v}$. A block can also contain other blocks. Defining a block with sub-blocks makes it modular and is easier to comprehend.

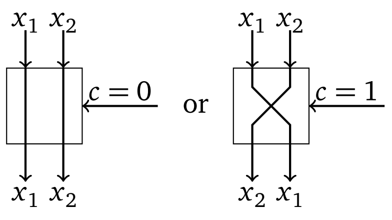
We measure a block with the so-called size and depth. The *size* of a block are the number of gates $C^k_{u,v}$ consists of (i.e. the size is k). We want to minimize the size by searching for a boolean circuit $C'^{k'}_{u,v}$ with k' =min$\{k : C^k_{u,v}$ computes $f\}$. In case of PFE, [KS08b] show that XOR gates (see Tab. 2.1a) can be evaluated for free. Therefore, we want to optimize the size with k' =min$\{k'' : C^k_{u,v}$ computes f and $k = k'' + \#$XOR gates$\}$. The *depth* of a block is measured by the number of gates in the longest possible path in $C^k_{u,v}$. Minimizing the size and the depth is important for a good performance. Depending on the used SFE protocol, the importance of size and depth difer: Yao [Yao82; Yao86] is evaluated gate by gate, so size is more important, while GMW [GMW87] is evaluated layer-by-layer, so depth is more important, since in the online phase there are number of AND depth communication rounds between the parties.

A *programmable block* $B^{P(n)}_{u,v}$ is a $B_{u,v}$ block with n additional (hidden) inputs. It can be programmed by providing a programming vector $c \in \{0,1\}^n$. An example of a programmable block is a UC. Every universal gate in a UC has to be programmed. We want to introduce two more important programmable blocks: Y-switching block and X-switching block.

A *Y-switching block* is a programmable block $B^{P(1)}_{2,1}$. It computes a function $f_Y : \{0,1\}^2 \times \{0,1\} \to \{0,1\}$ with $f_Y((x_1,x_2)^T, c) = x_{c+1}$ as visualized in Fig. 2.1a. So, the output of a Y-switching block is exactly one of its inputs depending on bit c. A Y-switching block provides the same functionality as a 2-input multiplexer. It can be programmed with 1 AND gate and 2 XOR gates (see Eq. 21 in [KS16]).



(a) Y Switching Block



(b) X Switching Block

Figure 2.1: Switching Blocks

An *X-switching block* is a programmable block $B_{2,2}^{P(1)}$. It computes a function $f_X : \{0,1\}^2 \times \{0,1\} \rightarrow \{0,1\}^2$ with $f_X((x_1, x_2)^T, c) = (x_{1+c}, x_{2-c})^T$ as we see in Fig. 2.1b. This means the inputs of an X-switching block are also the outputs of it, switched or not switched, depending again on bit c. An X-switching block can be programmed with 1 AND gate and 3 XOR gates (see Eq. 21 in [KS16]).

Y- and X-switching blocks are firstly introduced in [KS08a; KS08b].

2.4 Methods for SFE

In this section, we provide methods for solving the *Secure Function Evaluation (SFE)* problem. SFE describes the problem that two parties want to compute a function $f(x, y)$ with their secret inputs x and y and both want to learn $f(x, y)$ while keeping their inputs secret. There are two protocols that solve the problem using the boolean circuit representation of functions: *Yao's Garbled Circuit* [Yao82; Yao86] and the *GMW protocol* [GMW87]. We summarize both protocols in Sections 2.4.2 and 2.4.3. However, we first introduce *Oblivious Transfer (OT)* in Section 2.4.1 since both protocols rely on this primitive.

2.4.1 Oblivious Transfer

Oblivious Transfer (OT) is used for exchanging information between two parties - the sender and the receiver. In a 1-out-of-2 OT protocol, the sender has 2 messages m_0 and m_1 and the receiver can choose one of them by providing a bit $i \in \{0,1\}$. After this process, the receiver learns the message m_i (but not the other message) from the sender and the sender learns nothing, i.e. the sender does not know which message the receiver accesses. Oblivious transfer protocols rely on expensive public-key operations. Therefore, [IKNP03; ALSZ13; KOS15] provide OT extensions which enable the generation of any polynomial number of OTs using efficient symmetric key cryptography based on a number of base OTs.

2.4.2 Yao's Garbled Circuit Protocol

Yao's Garbled Circuit [Yao82; Yao86] allows SFE for two parties. One party is the function garbler, i.e. they know the boolean circuit description of the function $f(x, y)$ and their own input x while the so-called function evaluator only knows their input y, and the topology of the circuit representation of f.

Every gate in the boolean function is translated to a randomly encrypted gate, i.e. 0 and 1 are represented as random keys and the entries of the gates tables are randomly exchanged. The keys and the random gates tables are sent to the function evaluator who uses the 1-out-of-2 OT protocol to get the correct key depending on his input for each gate. After this computation, the function provider decrypts the outputs and send them to the function evaluator.

Many optimizations of Yao's garbled circuit protocol were proposed since its appearance, most relevant in our case is the free-XOR technique proposed in [KS08b], which enables XOR gates to be evaluated without communication effort.

2.4.3 GMW Protocol

The *GMW Protocol* [GMW87] is made for boolean circuits which only contain AND and XOR gates. It is based on secret sharing with the One-Time-Pad. So, all inputs and outputs of the gates are secret shared between two parties P_1 and P_2, i.e. each party secret shares his own input and gives a share to the other party. If P_1 has input a, he splits a into two shares $a = a_1 \oplus a_2$ (we denote with $\oplus$ the XOR operation) and sends a_2 to P_2. P_2 will not know the input a since he only has the seemingly random share.

The computation for the gates is done as follows: XOR gates with inputs a and b and output c can be evaluated with $c = c_1 \oplus c_2 = (a_1 \oplus a_2) \oplus (b_1 \oplus b_2) = (a_1 \oplus b_1) \oplus (a_2 \oplus b_2)$, i.e. each party P_i can compute his output share c_i locally with $c_i = a_i \oplus b_i$.

AND gates can be evaluated by using multiplication triples. A multiplication triple is a triple (x, y, z) with $(z_1 \oplus z_2) = (x_1 \oplus x_2)(y_1 \oplus y_2)$ [Bea91]. Both parties generate a random multiplication triple for each AND gate using 2 OT operations such that party i has the values x_i, y_i and z_i. This can be pre-computed in the offline phase. The next step is that party P_i sends the values $u_i = a_i \oplus x_i$ and $v_i = b_i \oplus y_i$ to the other party. We set $u = u_1 \oplus u_2$ and $v = v_1 \oplus v_2$. Both parties can now compute their part c_i of the result with $c_i = u \cdot y_i \oplus v \cdot x_i \oplus z_i \oplus (i-1) \cdot u \cdot v$ (we note that only party 2 uses the last XOR operation due to the $i-1$ multiplication).

3 Universal Circuit Constructions

In this chapter, we introduce the existing UC constructions. Therefore, we firstly introduce basic concepts for this topic in Section 3.1. Then, we explain the UC construction of Valiant [Val76] and the improvements of [LMS16] in Section 3.2. Another, more modular, UC construction provide [KS08a]. We shortly explain this construction in Section 3.3.

3.1 Basic Concepts

In this section we give definitions and basic concepts for Valiant's UC construction. This contains the definitions of a Universal Gate (UG) and a UC in Section 3.1.1 as well as the definition of EUGs in Section 3.1.2.

3.1.1 Universal Gates and Circuits

In this Section we extend the concepts of boolean gates and circuits presented in Section 2.1.

Representing the private function f as boolean circuit $C^k_{u,v}$ is important for the UC, since SFE methods such as Yao garbled circuits [Yao86] and GMW protocol [GMW87] highly rely on the circuit representation of the functionality. We are indeed able to represent concrete functions with boolean circuits but we do not gain any privacy so far. Since in PFE the function has to be private we have to find a way to make a boolean circuit private. Firstly, we want to provide Valiant's idea about UG. After that we explain and formalize UC.

The functionality of a G^2 gate is defined by the last column of its gate table (see Table 2.1). We can therefore use a vector $c \in \{0,1\}^4$ to define the implementation of it. For example, the XOR gate in Table 2.1a is defined by $c_{XOR} = (0,1,1,0)^T$. It is easy to see that we have 16 different G^2 implementations. Since the UC should support any of these 16 G^2 gates, Valiant introduces *Universal Gates (UGs)* [Val76] which are the implementation of a function $ug : \{0,1\}^2 \times \{0,1\}^4 \to \{0,1\}$. A UG, which shall compute an XOR gate, implements the function $ug(x, c_{XOR})$. We now have only one type of UG that can compute any of the 16 G^2 gates. The implementation table of a UG is shown in Table 3.1 and follows the Equation (3.1).

x_1	x_2	$ug(x,c)$
0	0	c_1
0	1	c_2
1	0	c_3
1	1	c_4

Table 3.1: Implementation table of a UG

$$ug(x,c) = \overline{x_1 x_2} c_1 + \overline{x_1} x_2 c_2 + x_1 \overline{x_2} c_3 + x_1 x_2 c_4 \tag{3.1}$$

Now, we are able to define, what a UC is. A *Universal Circuit* $UC^k_{u,v}$ has u inputs, v outputs and k distinguished UG. It can be programmed to implement every possible boolean circuit $C^{k_C \leq k}_{u,v}$. It is easy to see, that UG are the better solution for UC because of their dynamic programming of the control bits c. We will see more details about the construction of a UC in Sections 3.2 and 4.1.

The next step is to transform C_f to an input which the UC is able to read. With this transformation we can gain privacy of the function f and can potentially compute more than one function with the same UC. This process is discussed in Section 5.1.

3.1.2 Edge-Universal Graphs

This Section deals with Valiant's idea of an Edge-Universal Graph (EUG) [Val76]. We want to explain the characteristics of EUG and give a small example.

Let G_C be the Γ_2 graph of a boolean circuit $C^k_{u,v}$ and η a labelling on G_C (see Section 2.2). In case of PFE we need another Γ_2 graph uc to map G_C on uc due to the function privacy requirement. Valiant's solution is to edge-embed G_C to uc [Val76]. *Edge-embedding* is a mapping from graph $G = (V, E)$ into $G' = (V', E')$ with $V \subseteq V'$ and E' containing a path for each $e \in E$ in which the path are pairwise edge-disjoint. That means for every edge $e = (a_i, a_j) \in E$ exists a path $(a_i, \ldots, a_j)$ in G' and every $e' \in E'$ is in at most one of these paths.

A graph G' is an *Edge-Universal Graph* (EUG) if every graph $G \in \Gamma_2(n)$ can be edge-embedded into G'. Therefore, G' has distinguished *poles* $\{p_1, \ldots, p_n\} \subseteq V'$ where each node $a \in V$ is mapped to exactly one of these poles. This mapping is defined by the labelling η such that we can define a mapping $\varphi^V : V \to V'$ with $\varphi^V(a) = p_{\eta(a)}$. Now, we have to define for each edge $(a_i, a_j) \in E$ a path $(\varphi^V(a_i), \ldots, \varphi^V(a_j) = (p_{\eta(a_i)}, \ldots, p_{\eta(a_j)})$ in G' without using edges twice. We denote by $U_n(\Gamma_p)$ an EUG for $\Gamma_p(n)$ graphs with n poles.

Let $U_n(\Gamma_1) = (V, E) \in \Gamma_2$ be an EUG with poles $P = \{p_1, \ldots, p_n\}$. We can create an EUG $U_n(\Gamma_p)$ for each $p \geq 2$ by copying p times the $U_n(\Gamma_1)$ EUG, mapping each pole to one another and copy all edges and nodes which are no poles. Let $U_n(\Gamma_p) = (V', E') \in \Gamma_p$ an EUG constructed with p $U_n(\Gamma_1)$ graphs $(U_n(\Gamma_1)_1 = (V_1, E_1), \ldots, U_n(\Gamma_1)_p = (V_p, E_p))$. Than $V' = P \bigcup_{i=1}^{p} V_i \backslash P$ and $E' = \bigcup_{i=1}^{p} E_i$. This produces indeed a Γ_p graph because every pole has at most p inputs and outputs and every normal node has at most 2 inputs and outputs.

We give an example for better understanding. Let $G = (V, E)$ be the graph shown in Figure 3.1a. Our aim is to edge-embed G to an EUG $U_5(\Gamma_2)$. Therefore, we use two instances of the $U_5(\Gamma_1)$ EUG in Figure 3.1b. The two instances are shown in Figures 3.1c and 3.1d, denoted by $U_5(\Gamma_1)_1$ and $U_5(\Gamma_1)_2$. The edges $(a_1, a_4), (a_2, a_3)$ and (a_3, a_5) are edge-embedded in $U_5(\Gamma_1)_1$, the remaining edges are edge-embedded in $U_5(\Gamma_1)_2$. Merging $U_5(\Gamma_1)_1$ and $U_5(\Gamma_1)_2$ produces the wanted $U_5(\Gamma_2)$ graph shown in Figure 3.1e.

Valiant proposed to define *uc* such that it is an EUG. We will see, that it is a good strategy to edge-embed a Γ_2 graph with two instances of a $U_5(\Gamma_1)$ graph and merging them together. We use this trick in our implementation (Chapter 5).

3.2 Valiant's Universal Circuit

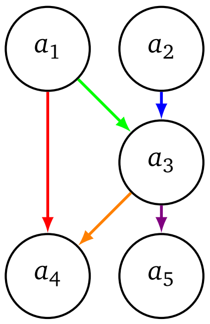
In this section, we want to describe Valiant's UC construction including his EUG constructions [Val76].

Valiant defined in [Val76] a UC which is able to evaluate every function of a certain size (see Section 3.1.1). We notice that the size n of a function f is given by the sum of its number of inputs u, outputs v and gates k (i.e. $n = u + v + k$). In the following, we describe step by step Valiant's way from a boolean circuit C_f to a UC construction which can be programmed to evaluate f. For that, we denote f for the function which shall be computed and u, v and k denote the number of inputs, outputs and gates of f. Additionally, $C_{u,v}^k$ denotes the corresponding boolean circuit which computes f. We describe this circuit as a graph $G_f \in \Gamma_2$ (see Section 2.2).

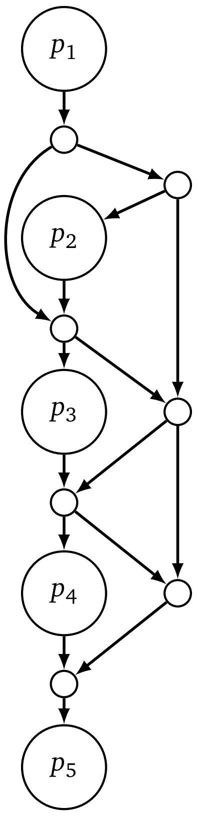
3.2.1 Overview

In this section, we describe the overall construction of Valiant's UC [Val76].

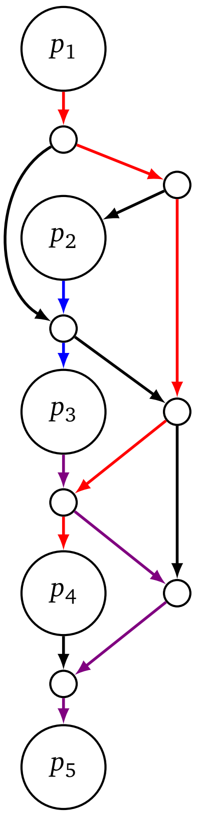
Valiant's UC is based on an EUG $U_n(\Gamma_2) = (V, E) \in \Gamma_2$. We notice that $U_n(\Gamma_2)$ can be also transformed to a boolean circuit. V has a subset P which are the poles of the $U_n(\Gamma_2)$ (see Section 3.1.2 for more details). These poles correspond to the inputs, outputs and gates of $C_{u,v}^k$, i.e. poles $1-u$ correspond to the inputs, poles $(u+1)-(u+k)$ to the gates and $(u+k+1)-n$ to the outputs. Valiant's aim is to create $U_n(\Gamma_2)$ such that it can be transformed to any boolean circuit which is able to compute functions with u inputs, v outputs and k gates.



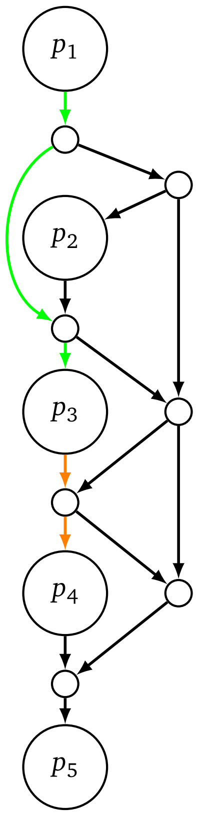
(a) $\Gamma_2(5)$ graph



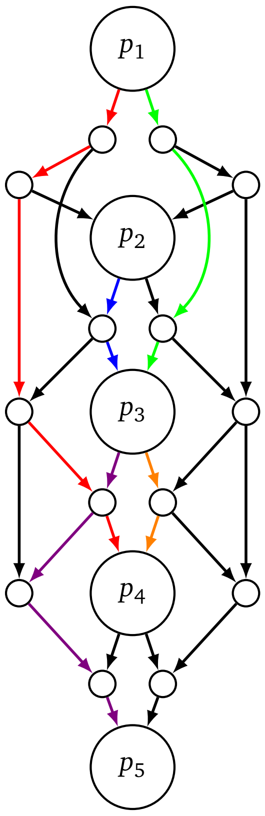
(b) EUG $U_5(\Gamma_1)$



(c) $U_5(\Gamma_1)_1$



(d) $U_5(\Gamma_1)_2$



(e) EUG $U_5(\Gamma_2)$

Figure 3.1: (a) shows a $\Gamma_2(5)$ graph. (b) shows an EUG $U_5(\Gamma_1)$ with 5 poles $(p_1, \ldots, p_5)$. (c)-(d) show the splitted edge-embedding of graph (a) having two $U_5(\Gamma_1)$ instances. (e) shows the edge-embedding of graph (a) with one $U_5(\Gamma_2)$ graph.

In the circuit $G_f = (V_f, E_f)$ which shall be programmed in $U_n(\Gamma_2)$, the gates' inputs can be outputs of other gates or inputs. Since we describe the UC as a Γ_2 graph, every gate can have at most two inputs and outputs. The restriction having at most two outputs is also valid for the inputs. So, G_f has to be transformed to cover these two restrictions. After this transformation, every node in G_f has indegree/outdegree at most 2. We describe this transformation in Section 2.1 for boolean circuits. Additionally, we need a labelling η on V_f in topological order, i.e. $\forall v_i, v_j \in V_f : \eta(v_i) > \eta(v_j) \Rightarrow$ there is no direct path between v_i and v_j in G_f.

Every node $v \in V$ is fulfilled with a unique task depending on their number of inputs and outputs as well as their node type (which means if they are poles or not):

- If v is a pole and corresponds to a gate in G_f, v is programmed as a UG (see Section 3.1.1).
- If v is no pole and has indegree 2 and outdegree 2, v is programmed as an X-switching block (see Figure 2.1b).
- If v is no pole and has indegree 2 and outdegree 1, v is programmed as a Y-switching block (see Figure 2.1a).
- If v is no pole and has indegree 1 and outdegree 2, v is programmed as a copy gate, i.e. the input of v is copied to both outputs.
- If v is no pole and has indegree 1 and outdegree 1, v should be removed since the input and output of v are equal.

We see, nearly every node in G_f (except for input and output poles) are another small boolean circuit. The construction of these boolean circuits are not part of this work since our tool does not need them. Therefore, we refer to [KS16; LMS16] who provide optimal constructions. We recapitulate the sizes of these constructions in Chapter 6, which, however, affects the size of our resulting UC as well.

For a better abstraction, we consider the nodes programmed as UG, X-switching block or Y-switching block as programmable blocks. So, we need for each of those nodes a programming vector c depending on the function f. To determine the exact programming of the nodes and thus of the UC we have to edge-embed $U_n(\Gamma_2)$ into G_f. This edge-embedding is a difficult problem for general Γ_2 graphs. Therefore, we have to develop more efficient methods which are optimized for the considered EUG construction.

Currently, it is easier to edge-embed Γ_1 graphs instead of Γ_2 graphs. Therefore, we split $U_n(\Gamma_2)$ into two instances $U_n(\Gamma_1)_1$ and $U_n(\Gamma_1)_2$ which can be edge-embedded seperately and afterwards get merged to the final $U_n(\Gamma_2)$ UC. This also means, that G_f has to be splitted into two Γ_1 graphs. [KS16] provide a method for this which can also be used for the edge-embedding, in our case with small modifications. We detail this method in Section 4.2.2.

Now, we want to introduce the two EUG constructions Valiant provides.

3.2.2 Valiant's EUG Constructions

Valiant provides two EUG constructions [Val76] which are the base of the UC construction as described in the previous section. In this section we want to describe these two constructions and give an overview of [LMS16]'s generalization of Valiant's constructions.

We described in Section 3.1.2 that a $U_n(\Gamma_f)$ EUG can be constructed of f $U_n(\Gamma_1)$ EUG. Therefore, Valiant provides EUG for Γ_1 graph which can be extended to EUG for Γ_2 graphs. In general, the graph of a $U_n(\Gamma_1)$ construction depends on its number of poles $|P| = n$. Let $P = \{p_1, \ldots, p_n\}$ be the set of poles $U_n(\Gamma_1)$ consists of and having indegree and outdegree 1. Then, the first EUG construction by Valiant is shown in Figure 3.2. We emphasize the poles as big circles and the additional nodes needed to make $U_n(\Gamma_1)$ universal are emphasized as small circles or squares. The squares are special nodes since they are the key nodes for the recursive construction. Let $Q_{\lceil \frac{n-2}{2} \rceil} = \{q_1, \ldots, q_{\lceil \frac{n-2}{2} \rceil}\}$ and $R_{\lceil \frac{n-2}{2} \rceil} = \{r_1, \ldots r_{\lceil \frac{n-2}{2} \rceil}\}$. The two sets $Q_{\lceil \frac{n-2}{2} \rceil}$ and $R_{\lceil \frac{n-2}{2} \rceil}$ are the poles of the next recursion step. With these poles, we build another EUG (also called subgraphs) which produce new sets $Q_{\lceil \frac{\lceil \frac{n-2}{2} \rceil -2}{2} \rceil}$ and $R_{\lceil \frac{\lceil \frac{n-2}{2} \rceil -2}{2} \rceil}$, respectively, i.e. we have 4 of those sets at this level. We notice that these two sets are different nodes for $Q_{\lceil \frac{n-2}{2} \rceil}$ and $R_{\lceil \frac{n-2}{2} \rceil}$. The recursion base is reached if the number of poles is between 1 and 4. U_1 is one single node (pole), U_2 are two connected poles, U_3 is a graph in which poles 1 and 2 are connected and poles 2 and 3 are connected. U_4 is constructed with 3 additional nodes which are between 2 poles, i.e. there is alternately a pole and a node from top to bottom. Valiant also provides EUG constructions for U_5 and U_6 [Val76]. [Val76; KS16] prove that this construction is a valid EUG construction. This construction is called the *2-way split EUG construction* since there are two sets of recursion nodes. [KS16] provide an implementation of a UC using this 2-way split construction optimized for PFE application. At the same time, [LMS16] also implement a UC based on this EUG construction.

Valiant provides another EUG construction, called the *4-way split EUG construction*. This construction has smaller size and is not used for a UC implementation so far. Therefore, we improve the implementation of [KS16] by using the 4-way split EUG construction. We will detail this EUG construction in Section 4.1.

The EUG constructions show that for every number of inputs u, outputs v and gates k, there exist a UC constructed by a graph of two EUG. This construction has complexity $\mathcal{O}(k \log k)$, which is proven to be the asymptotically optimal size in [Weg87].

Lipmaa et al. provide an approach for multiple EUG constructions, the so-called *k-way split EUG construction* [LMS16]. So, the construction depends on the value k. For $k = 2$ and $k = 4$ it makes sense to use the asymptotically optimized EUG constructions of Valiant [Val76]. For other k, [LMS16] use parts of the UC construction of Kolesnikov and Schneider in [KS08a] which we shortly introduce in 3.3.

The idea is to split $n = u + v + k$ in $m = \lceil \frac{n}{k} \rceil$ blocks. Every block i consist of k inputs $r_{i-1}^1, r_{i-1}^2 \ldots r_{i-1}^k$ and k outputs $r_i^1, r_i^2 \ldots r_i^k$ as well as k poles (except of the last block which has $n \bmod k$ poles). For every $j \leq k$, the list of all r_i^j build a next recursion step, i.e. we

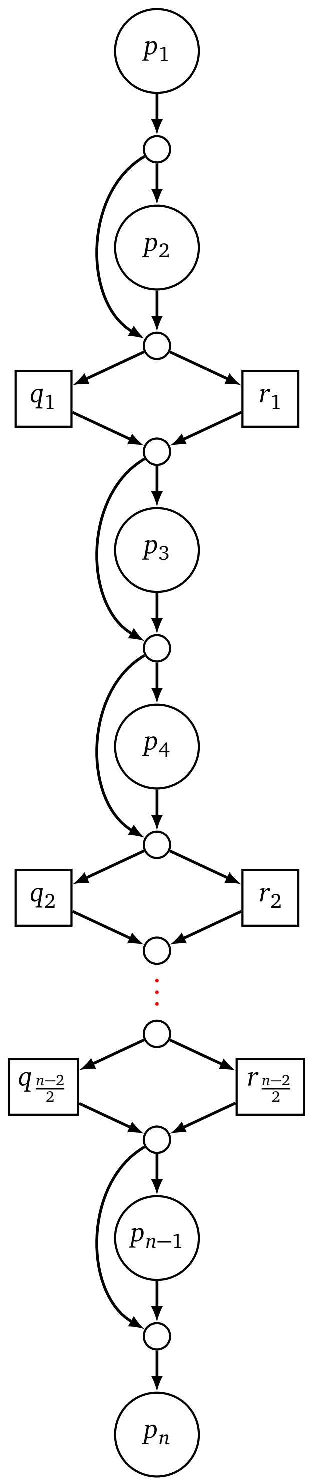


Figure 3.2: Valiant's 2-way EUG [Val76]

have k recursion steps similar to Valiant's 2-way split EUG construction. Additionally, every block begins and ends with a permutation network such that the inputs and outputs can be permutated to every pole. Therefore, they put a Y-switching block in front of every pole p_i which is connected to the $i-th$ output of the permutation network as well as the $i-th$ output of a block-intern EUG $U_k(\Gamma_1)$. That means, we reduce the problem to find an EUG $U_n(\Gamma_1)$ to the problem to find an EUG $U_k(\Gamma_1)$. Their solution is to build the block-intern EUG with components of the UC construction of [KS08a].

However, [LMS16] optimize the construction for the number of total gates without considering of free XOR-gates evaluation [KS08b].

3.3 A Modular Universal Circuit Construction

There exists another UC construction provided by [KS08a]. Their construction is more modular than Valiant's constructions. However, this construction has a deeper depth and the size is asymptotically worse which is also shown in [KS08a]. In this section, we briefly summarize this construction.

Let u, v and k denote the number of inputs, outputs and gates as before. [KS08a] designed their UC construction with three programmable blocks: two different selection blocks and one universal block. Those blocks hide the wiring of the inputs and outputs.

A *Selection Block* S_z^p is a programmable block that has p inputs and z outputs. The inputs are mapped to the outputs in any order and any frequency. That means, it is possible that one input is forwarded to any output or one input does not occur in any output. [KS08a] provide an efficient selection block construction with size $\mathcal{O}(k \log k)$ and depth $\mathcal{O}(k)$.

The *Universal Block* U_n is also a programmable block with $2n$ inputs and n outputs. This block simulates n universal gates (UG) and makes any wiring of a fanin 2 acyclic circuit possible. Therefore, the circuit needs again a labelling η in topological order as mentioned in Section 3.2.2 such that no output of a gate with labelling i can be an input of another gate with labelling $j < i$. Kolesnikov and Schneider in [KS08a] construct their universal block recursively using two smaller universal blocks, a selection block and one mixing block, which has two sets of inputs $(in_1^1, \dots, in_n^1)$ and $(in_i^2, \dots, in_n^2)$ and n outputs with $out_i = in_i^{c_i}$ (we note that c is the programming vector of the mixing block). The depth of their universal block is $\mathcal{O}(k \log k)$ and the size is $\mathcal{O}(k \log^2 k)$, i.e. their construction does not achieve the asymptotically optimal size of $\mathcal{O}(k \log k)$.

Having described the building blocks of the UC construction of [KS08a], we now describe how these programmable blocks build up a universal circuit. The u inputs of the UC are the inputs of a $S_{2k \leq u}^u$ selection block which shall hide the wiring of the inputs by directing them obliviously into the most possible input number $(2k)$ of wires. The $2k$ outputs of this block are forwarded to a U_k universal block, which simulates the k gates as mentioned above. The k outputs of this block are forwarded to a $S_v^{k \geq v}$ selection block, which shall hide the wiring of the outputs by obliviously choosing from the maximum available k output wires the v outputs used by the actual circuit. [KS08a] provide customized switching blocks which have smaller size.

3.3.1 Hybrid Constructions

In this section we explain shortly the hybrid UC constructions of Kiss and Schneider in [KS16].

[KS16] design a UC construction using both approaches of Valiant's construction in [Val76] and the [KS08a] UC construction. The universal block of [KS08a]'s construction makes the whole construction asymptotically large. Instead of using a universal block, one can use Valiant's UC between the selection blocks.

In case of large number of inputs $u \sim k$ but constant number of outputs, they propose to use only the first selection block while in case of large number of inputs $u \sim 2k$ and large number of outputs $v \sim k$, the construction with both selection block might be the better one. However, in any other case Valiant's construction used by its own provides a better size and depth. We note that the most common case is a small (constant) number of inputs and outputs, for which Valiant's construction performs best. More details can be found in [KS16].

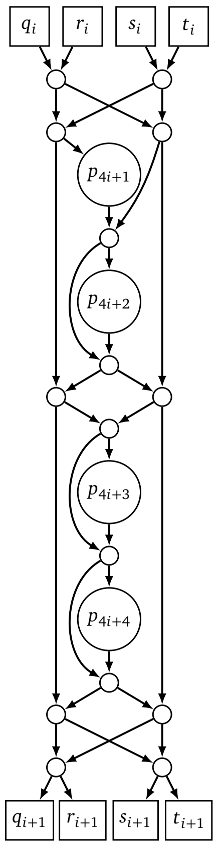
4 Making Valiant's EUG Construction Modular

In addition to his 2-way split EUG construction, Valiant also offers a more efficient 4-way split EUG construction [Val76]. Lipmaa et al. extend Valiant's work by considering a k-way split EUG constructions in [LMS16]. We improve their work by making Valiant's k-way split EUG construction more modular at the example of $k = 4$. This enables us to realize this construction efficiently in practice, which is detailed in Chapter 5. Therefore, we shortly describe Valiant's 4-way split EUG construction with our optimizations in Section 4.1. Thereafter, we provide our approaches for splitting the edge-embedding task into two main parts with the help of the supergraph construction from [KS16] in Section 4.2. We finish this chapter with giving an approach for the edge-embedding of a k-way split construction which is also based on [KS16]'s considerations in Section 4.2.4.

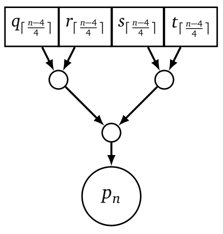
4.1 Valiant's 4-way Split EUG Construction

Similar to his 2-way split EUG construction (meaning that the recursion using 2 smaller instances), Valiant also provides a 4-way split EUG construction for Γ_1 graphs which can be extended to a construction for Γ_2 graphs by utilizing two instances $U_n(\Gamma_1)_1$ and $U_n(\Gamma_1)_2$ as before (see Section 3.2). The construction with our optimizations is visualized in Figure 4.1. Valiant offers the main, so-called *Body block* (see Figure 4.1a) as programmable block consisting of 4 poles (emphasized as big circles), 15 nodes (emphasized as small circles) as well as 8 recursion points (emphasized as squares). These body blocks are built up together such that the 4 top [bottom] recursion points of one block are the 4 bottom [top] recursion points of the other block. Similar to the 2-way EUG construction, we create 4 sets for a graph with n nodes, i.e., $Q_{\lceil \frac{n-4}{4} \rceil} = \{q_1, \ldots, q_{\lceil \frac{n-4}{4} \rceil}\}$, $R_{\lceil \frac{n-4}{4} \rceil} = \{r_1, \ldots, r_{\lceil \frac{n-4}{4} \rceil}\}$, $S_{\lceil \frac{n-4}{4} \rceil} = \{s_1, \ldots, s_{\lceil \frac{n-4}{4} \rceil}\}$ and $T_{\lceil \frac{n-4}{4} \rceil} = \{t_1, \ldots, t_{\lceil \frac{n-4}{4} \rceil}\}$ which are the poles of the next recursion step, respectively. This means, for every recursion step, we get 4 subgraphs which also create 4 subgraphs until we reach the recursion base (this is the case if the number of recursion points is less than or equal to 4, see Section 3.2.2).

We note that the top [bottom] block does not need the upper [lower] recursion points since those poles are w.l.o.g. the real inputs [outputs]. Therefore, we design special blocks - called *Head* and *Tail Block*. A *Head Block* is a programmable block $B_{0,k}^{P(l)}$ which only occurs at the top of a chain of blocks. In case of the 4-way split EUG construction, it is a $B_{0,4}^{P(l)}$ programmable block and consists of 4 poles, 10 nodes and 4 recursion points at the bottom (therefore $k = 4$ in this case) as visualized in Figure 4.1f. We can determine the number of additional hidden

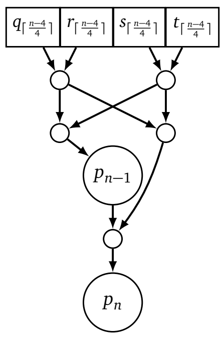


(a) Body Block



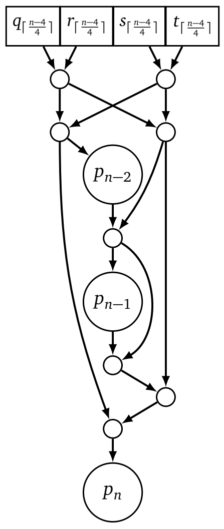
$q_{\lceil \frac{n-4}{4} \rceil}$ $r_{\lceil \frac{n-4}{4} \rceil}$ $s_{\lceil \frac{n-4}{4} \rceil}$ $t_{\lceil \frac{n-4}{4} \rceil}$
p_n

(b) Tail Block (1)



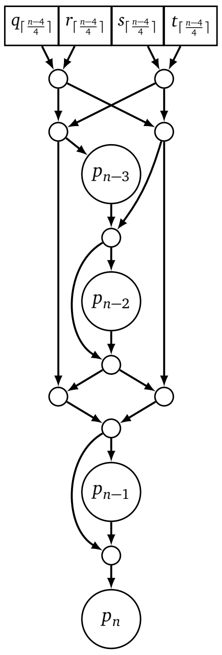
$q_{\lceil \frac{n-4}{4} \rceil}$ $r_{\lceil \frac{n-4}{4} \rceil}$ $s_{\lceil \frac{n-4}{4} \rceil}$ $t_{\lceil \frac{n-4}{4} \rceil}$
p_{n-1}
p_n

(c) Tail Block (2)



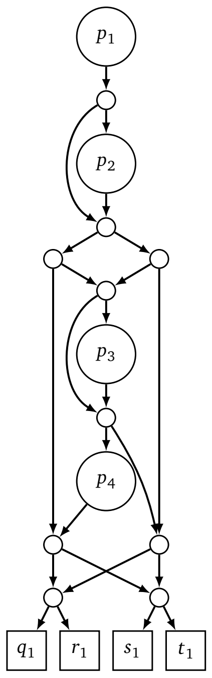
$q_{\lceil \frac{n-4}{4} \rceil}$ $r_{\lceil \frac{n-4}{4} \rceil}$ $s_{\lceil \frac{n-4}{4} \rceil}$ $t_{\lceil \frac{n-4}{4} \rceil}$
p_{n-2}
p_{n-1}
p_n

(d) Tail Block (3)



$q_{\lceil \frac{n-4}{4} \rceil}$ $r_{\lceil \frac{n-4}{4} \rceil}$ $s_{\lceil \frac{n-4}{4} \rceil}$ $t_{\lceil \frac{n-4}{4} \rceil}$
p_{n-3}
p_{n-2}
p_{n-1}
p_n

(e) Tail Block (4)



(f) Head Block

Figure 4.1: (a) shows Valiant's 4-way split EUG construction [Val76]. (b)-(e) show our tail block constructions for different number of poles (denoted in brackets). (f) shows our head block construction.

inputs as $l = 7 + 4 \cdot \{\text{number of UG in this block}\}$, since we have 1 additional control bit for each X-switching block as well as 4 control bits for each UG.

As a counterpart, we define *Tail Blocks* as programmable blocks $B_{k,0}^{P(l)}$ which only occur at the bottom of a chain of blocks. In our case, we have a $B_{4,0}^{P(l)}$ programmable block. It consists of less than or equal to 4 poles (more precisely the remaining number of poles at the end of the chain $(n \bmod 4) + 1$), 4 recursion points at the top and less than 11 nodes depending on the number of poles. The 4 different tail block constructions are visualized in Figures 4.1b, 4.1c, 4.1d and 4.1e. The value of l depends also on the number of poles in the block: $l = \{\text{ number of nodes without poles }\} + 4 \cdot \{\text{ number of UG in this block }\}$.

We notice that the 4 recursion base constructions are the remaining blocks for this EUG construction as mentioned in Section 3.2.2.

Depending on the number of inputs and outputs of the nodes and recursion points, the nodes are the implementation of various gates or blocks. The respective gates and blocks can be found in Section 3.2.1.

4.2 Edge-Embedding

In this section we give approaches for handling the edge-embedding on Valiant's EUG. First, we explain the edge-embedding on the example of Valiant's 4-way split EUG construction [Val76] and in Section 4.2.4 we provide an approach for the edge-embedding of k-way split EUG constructions.

The edge-embedding can be split into two main parts: the block edge-embedding and the recursion point edge-embedding. The idea of the block edge-embedding is that every block handles their edge-embedding task themselves. We detail this in Section 4.2.1.

It is easy to see that only the recursion points are left while every block handles the programming of the nodes itself. Therefore, we consider the recursion point edge-embedding which sets the programming bit of the remaining recursion points. We use the supergraph construction of [KS16] to handle this task in Section 4.2.2.

The last step is to combine both edge-embedding types in Section 4.2.3. This is important since the single blocks have to know how to edge-embed themselves.

4.2.1 Block Edge-Embedding

The task of the block edge-embedding is to manage the edge-embedding inside of a block, i.e. every block edge-embeds their nodes within themselves. Therefore, a block does not need to consider the other block's programming.

In our case we consider the 4 top [bottom] recursion points of the block as intermediate nodes where the inputs [outputs] of the block go through (note: a head block has only outputs and a tail block only inputs). Valiant's idea is that any of these inputs can be forwarded to exactly one of the 4 poles of the block and any pole can be forwarded to exactly one output or another pole having a higher topological labelling value.

We formalize this behaviour in the following: Let $B_{4,4}^{P(19)}$ be the programmable block visualized in Figure 4.1a with poles $p_1, \dots, p_4$ and η a topological ordered labelling of the nodes (and poles). The nodes $r_0^1, \dots, r_0^4$ and $r_1^1, \dots, r_1^4$ denote the input and output recursion points of B. Additionally, $in = (in_1, \dots, in_4) \in \{0, \dots, 4\}^4$ and $(out_1, \dots, out_4) \in \{0, \dots, 7\}^4$ denote the input and output vectors of B. The value 0 of the input and output vectors is the so-called *Dummy value* which is used if an input [a pole] is not forwarded to any pole [output] of B. The output vector has a larger value range due to the larger path options. A pole can be forwarded to another pole or an output recursion point. Therefore, we use the values 1, 2 and 3 for the poles p_2, p_3 and p_4 and the values 4, 5, 6 and 7 for the output recursion points. We notice that the pole p_1 can not be a destination for a path in B having another pole in B as input since $\eta(p_1)$ is less than the labelling of any other pole in B. Additionally, the values of in and out are pairise different or 0. The rules for the input and output vectors are formalized in Equation (4.1). A pair $(r_0^i, p_j) \in \mathcal{P}$ or $(p_i, r_1^i) \in \mathcal{P}$ denotes a path from r_0^i to p_j or p_i to r_1^i in the set of all paths $\mathcal{P}$ in B.

$$\begin{aligned}
\forall i \in \{1, \dots, 4\} : in_i \neq 0 \rightarrow (r_0^i, p_{in_i}) \in \mathcal{P} \\
\forall i \in \{1, \dots, 4\} : out_i \neq 0 \wedge out_i < 4 \rightarrow (p_i, p_{1+out_i}) \in \mathcal{P} \wedge \eta(p_i) < \eta(p_{1+out_i}) \\
\forall i \in \{1, \dots, 4\} : out_i > 3 \rightarrow (p_i, r_1^{i-3}) \in \mathcal{P} \\
\\
\forall in_i, in_j \in in : i \neq j \rightarrow in_i = 0 \vee in_i \neq in_j \\
\forall out_i, out_j \in out : i \neq j \rightarrow out_i = 0 \vee out_i \neq out_j
\end{aligned} \tag{4.1}$$

So, every combination of input and output vector covering these conditions are valid for B. We use this observation in the edge-embedding process described in the next section. In Section 5.2.1, we provide methods for solving the edge-embedding within the block.

4.2.2 Recursion Point Edge-Embedding

The block edge-embedding covers the programming of every node except for the recursion points. So, the task of the recursion point edge-embedding is to program the remaining recursion point nodes. Additionally, the second task of the recursion point edge-embedding is to set the input and output vectors of the blocks for the block edge-embedding. Therefore, we use the supergraph construction of [KS16] which split a $\Gamma_2(n)$ graph in two $\Gamma_1(n)$. These Γ_1 graphs are merged again to one $\Gamma_2(\lceil \frac{n}{2} \rceil)$ and so on. We now explain the single steps in the following. In the next section, we detail how to solve the two tasks of the recursion point edge-embedding, so this section only describes in detail the supergraph construction since this is the most important construction for the edge-embedding task.

Firstly, we notice that [KS16] use this construction for Valiant's 2-way split EUG construction [Val76]. However, we can modify the algorithm so that it also works for the 4-way split EUG construction.

Let $C^k_{u,v}$ be the boolean function f which our UC shall compute. $G \in \Gamma_2(u+v+k)$ denotes the transformed graph of C_f (see Section 2.2).

1. Splitting $G \in \Gamma_2$ in two Γ_1 graphs G_1 and G_2: Valiant's UC consists of the merging of two EUG for Γ_1 graphs. Therefore, we have to split G into two Γ_1 graphs G_1 and G_2, meaning that every node $v \in G_1$ and every node $w \in G_2$ has indegree and outdegree at most 1. With this splitting we get two circuits $C'^{k,1}_{u,v}$ which can be edge-embedded in an EUG for Γ_1 graphs. Merging both of the EUG for Γ_1 graphs results in the wanted UC which computes f.

The splitting of G can be done with 2-coloring G [Val76; KS16]. 2-coloring defines the following problem: Let $K = (V, E)$ be a directed graph. K can be 2-colored if we can create 2 sets E_1 and E_2 with the following conditions:

$$\begin{aligned} \forall e \in E &: (e \in E_1 \lor e \in E_2) \land \neg(e \in E_1 \land e \in E_2) \\ \forall e = (v_1, v_2) \in E_1 &: \neg\exists e' = (v_3, v_4) \in E_1 : v_2 = v_4 \lor v_1 = v_3 \\ \forall e = (v_1, v_2) \in E_2 &: \neg\exists e' = (v_3, v_4) \in E_2 : v_2 = v_4 \lor v_1 = v_3 \end{aligned} \tag{4.2}$$

We call the edges in E_1 the *blue edges* and the ones in E_2 the *black edges*. With these two sets E_1 and E_2 we can build the two Γ_1 graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$.

In case of Γ_2 graphs we can always find a 2-coloring [LP09; Val76]. [KS16] describe an algorithm for determining a valid 2-coloring with help of the proof of the König-Hall theorem in [LP09].

2. Merging one $\Gamma_1(n)$ graph into one $\Gamma_2(\lceil \frac{n}{2} \rceil)$ graph: The number of poles decreases in every recursion step. Therefore, we have to merge a $\Gamma_1(n)$ graph into a $\Gamma_2(\lceil \frac{n}{2} \rceil)$ graph. It has to be a Γ_2 graph in order to loose no information.

Let $G_1 = (V, E) \in \Gamma_1(n)$ be a topological ordered graph and $G_m = (V', E') \in \Gamma_2(\lceil \frac{n}{2} \rceil)$ a graph with nodes $v'_1, \ldots, v'_{\lceil \frac{n}{2} \rceil}$. We define two labellings η_{in} and η_{out} on G_m with $\eta_{\text{in}}(v_i) = i$ and $\eta_{\text{out}}(v_i) = \eta_{\text{in}}(v_i) - 1 = i - 1$. Additionally, we define a mapping θ_V that maps a node $v_i \in V$ to a node $v_j \in V'$ with $\theta(v_i) = v'_{\lceil \frac{i}{2} \rceil}$. That means two nodes in G_1 build one node in G_m. At last, we define a mapping θ_E that maps an edge $e_i = (v_i, v_j) \in E$ to an edge $e_j \in E'$ with $\theta_E((v_i, v_j)) = (v_{\eta_{\text{in}}(\theta_V(v_i))}, v_{\eta_{\text{out}}(\theta_V(v_j))})$. That means every edge in G_1 is mapped to one edge in G_m as follows: Let $e = (v_i, v_j) \in E$ and $e' = (v'_k, v'_l) \in E'$. Then it is $v'_k = \theta_V(v_i)$ that means v'_k is the new node of v_i in G_m. The node v_j differs. Here, v'_l is not the new node of v_j in G_m but it is v'_{l+1} (i.e. it is one less).

So, we can build the graph $G_m = (V', E')$ as follows: We set $V = \{v'_1, \dots, v'_{\lceil \frac{n}{2} \rceil}\}$ and $E' = \bigcup_{e \in E} \theta_E(e)$. Then we check for all $e = (v'_i, v'_j) \in E'$ whether $j < i$ and remove those edges in E'.

We now explain the construction with an example. Therefore, we take a look at Figure 3.2. Graph G_1 consists of the nodes $V = \{p_1, \dots, p_n\}$ and G_m consists of the nodes $V' = \{q_1, \dots, q_{\lceil \frac{n}{2} \rceil}\}$. Let $E = \{(p_1, p_3), (p_2, p_n), (p_3, p_4)\}$. The intension of G_m is to provide the paths of the next recursion step. That means the edge (p_1, p_3) should be mapped to the edge (q_1, q_1) since the path from p_1 to p_3 goes over q_1 in the next recursion step and leaves it also over q_1. It is $\theta_V(p_1) = q_1$ and $\theta_V(p_3) = q_2$. Here we see, why it is important to reduce the topological ordered labelling by 1 since the path from p_1 to p_3 does not go through q_2 but q_1.

The reason why we delete those edges $(q_i, q_j) \in E'$ with $j < i$ can be shown at the example of the edge (p_3, p_4). The path from p_3 to p_4 does not go through the next recursion step since it can be edge-embedded within the block. However, $\theta_E((p_3, p_4)) = (q_2, q_1)$ since q_2 is the new node of p_3 and p_4. Deleting them solves the problem and we do not have to farther consider them. It is surprising that there exists a node $q_{\lceil \frac{n}{2} \rceil}$ in V' but there is no such node in Valiant's 2-way split EUG construction [Val76]. However, there is no incoming nor outgoing edge so it can be deleted. We let it there since we need this node for our modification for Valiant's 4-way split EUG construction [Val76] what we explain later in this section.

3. Creating the supergraph construction of [KS16]: As mentioned above, we start with a graph $G \in \Gamma_2(n)$ which includes all the pole paths in the two EUGs. The first step is to split G into two Γ_1 graphs G_1 and G_2 where G_1 contains all the paths for the first EUG and G_2 contains the remaining paths for the second EUG. We now explain the following steps using the example of G_1 (for G_2 it is the same).

G_1 is merged to a Γ_2 graph G_{m_1} (m_1 denotes that it is the merged Γ_2 graph of G_1). Now, G_{m_1} get two-colored and creates the graphs G_1^{l} and G_1^{r}. These two graphs get merged two Γ_2 graphs $G_{m_1}^{\mathrm{l}}$ and $G_{m_1}^{\mathrm{r}}$. We denote with a pattern of "l" and "r" the positions of the subgraphs, meaning that G_1^{l} is the left subgraph of G_1 or G_1^{r} the right subgraph of G_1. Or recursively spoken: Let ψ be a pattern of "l" and "r" (e.g. $\psi = \mathrm{lrrlr}$). Then $G_1^{\psi \circ \mathrm{l}}$ denotes the left subgraph of G_1^{ψ}.

We continue these steps (merging a Γ_1 graph to a Γ_2 graph and splitting this in two Γ_1 graphs) until the resulting Γ_1 graph has 4 or less nodes (this is the recursion base) or does not contain any edges.

4. Modifications for Valiant's 4-way split EUG construction [Val76]: In Valiant's 4-way split EUG construction [Val76], we need a supergraph which creates 4 subgraphs in each step. Therefore, we need a 4-coloring after merging a Γ_1 graph to a Γ_4 graph where 4 nodes build a new node (as mentioned above). This problem can be directly solved by using the already mentioned 2-coloring. The supergraph construction of [KS16] provides after 2 times 2-coloring 4 subgraphs (G_1^{ll}, G_1^{lr}, G_1^{rl} and G_1^{rr}). We can use these graphs as the result of the 4-coloring.

However, there is one aspect to mention: the first 2-coloring is a preprocessing step, i.e. it does not map to an EUG recursion step. Therefore, we have to define another labelling η_{out_P} with $\eta_{\mathrm{out}_P}(v) = \eta_{\mathrm{in}}(v)$ (we note that here we indeed need the node $q_{\lceil \frac{n}{2} \rceil}$). Then the creation of the supergraph works as follow: We merge G_1 to a Γ_2 graph with labelling η_{out_P} and get G_{1_m}. After that we split G_{1_m} into two Γ_1 graphs G_1^{l} and G_1^{r}. These two graphs get merged to the Γ_2 graphs $G_{1_m}^{\mathrm{l}}$ and $G_{1_m}^{\mathrm{r}}$ using the η_{out} labelling. Finally, the graphs get splitted into the 4 Γ_1 graphs G_1^{ll}, G_1^{lr}, G_1^{rl} and G_1^{rr}. These are the important graphs for the first recursion step in Valiant's 4-way split EUG construction [Val76]. Now we start over for all 4 subgraphs until we reach 4 or less nodes in the Γ_1 graphs or there are no edges left (not in the preprocessing step).

4.2.3 Combining both Edge-Embedding Parts

In this section, we combine the block edge-embedding and the recursion point edge-embedding task to fulfil the whole edge-embedding. We explain this with Listing 4.1.

Let G denote the part of the EUG without recursion steps (i.e. G only consists of the main chain of blocks) and $G_1 = (V, E)$ denotes the corresponding Γ_1 graph which is one of the two graphs after the splitting process as described in the previous section. With these two graphs we are able to edge-embed G_1 into G.

We denote with S the set of the 4 subgraphs of G_1 in the supergraph, i.e. $S = \{G_1^{\mathrm{ll}}, G_1^{\mathrm{lr}}, G_1^{\mathrm{rl}}, G_1^{\mathrm{rr}}\}$ in the first function call. A recursion step graph of G is one of the generated graphs having one of the 4 sets of recursion points (e.g. $q_1, \dots, q_{\lceil \frac{n-4}{4} \rceil}$) as poles without generating the recursion steps (i.e. the recursion step graph is the chain of blocks which build the set of recursion points $Q_{\lceil \frac{n-4}{4} \rceil}$ (and for the other 3 sets respectively)). R denotes the set of all 4 recursion step graphs of G. Finally, let B be the set of all blocks in G.

We now want to explain lines 5 - 29 in Listing 4.1. Let $e = (v_i, v_j) \in E$ be any edge in G_1. b_i and b_j in lines 7 and 8 denote the blocknumbers in which v_i and v_j are. i' and j' denote the pole positions in the blocks B_{b_i} and B_{b_j}, i.e. $i' = ((i-1) \bmod 4) + 1$. out and r^1 [in and r^0] denote the output [input] vector and the corresponding set of output [input] recursion points as described in Section 4.2.1. We have to distinguish between 2 cases:

Case 1: v_i and v_j are in the same block

Listing 4.1: Edge-Embedding algorithm for Valiant's 4-way split EUG Construction [Val76]

```
procedure edge—embedding ($G$, $G_1 = (V, E)$)
  Let $S$ be the set of the 4 $\Gamma_1$ subgraphs of $G_1$ in the supergraph
  Let $R$ be the 4 recursion step graphs
  Let $B$ be the set of blocks in $G$
  $\forall e = (v_i, v_j) \in E$ do
        Let $i'$ and $j'$ denote the positions of $v_i$ and $v_j$ in their
           ↪ blocks
        $b_i \leftarrow \lceil \frac{i}{4} \rceil$
        $b_j \leftarrow \lceil \frac{j}{4} \rceil$
        Let $out$ [$r_1$] denotes the output vector [recursion points]
           ↪ of $B_{b_i}$
        Let $in$ [$r_0$] denotes the the input vector [recursion points]
           ↪  of $B_{b_j}$
        if $b_i = b_j$ do
          if $v_i \neq v_j$ do
            $out_{i'} \leftarrow j' - 1$
          end if
        else
          Let $s = (V', E') \in S$ denote the $\Gamma_1$ graph with $e' = (p_{b_i}, p_{b_{j-1}}) \in E'$
             ↪ and $e'$ is not marked
          Mark $e'$
          Let $x$ denote the number with $s = S_x$
          Set the control bit of $r_0^x$ to 1
          if $b_j = b_i$ + 1 do
            $y \leftarrow$ 0
          else
            $y \leftarrow$ 1
          end if
          Set the control bit of $r_1^x$ to $y$
          $out_{i'} \leftarrow x + 4$
          $in_x \leftarrow j'$
        end if
  end $\forall$
  Edge—embed all blocks in $G$
  $\forall i \in \{1, \ldots, 4\}$ do
        if $S_i$ exists do
      call edge—embedding($R_i$, $S_i$)
    end if
  end $\forall$
end procedure
```

That means, $b_i = b_j$. In this case, the edge-embedding can be solved within the block and no recursion points have to be programmed for this path. Therefore, we set the *out* vector of block B_{b_i} at position i' (which denotes the position of v_i in B_{b_i}) to $i' - 1$ (see Section 4.2.1).

Case 2: v_i and v_j are in different blocks

That means, $b_i \neq b_j$. We have to find an edge $e' = (p_{b_i}, p_{b_{j-1}})$ in one of the 4 subgraphs of G_1. This means the path in the next recursion step has to be between the poles p_{b_i} and $p_{b_{j-1}}$ (see Section 4.2.2). We denote with $s \in S$ the Γ_1 subgraph of G_1 which consists of such an unmarked e' (if an edge is marked, it is already used and cannot be reused for this task, so we have to search in an other subgraph). After we select e', we mark this edge so that it cannot be used any more in further course of the algorithm. We set x to the number of the subgraph s in S, i.e. $S_x = s$. This is important since we have to know in which of the 4 recursion step graphs we have to edge-embed a path from p_{b_i} to $p_{b_{j-1}}$. So, it also tells us which of the 4 recursion points we have to program. We can firstly set the programming bit of the x-th input recursion point r_0^x to 1 since the path between the poles with labelling i and j leaves the next recursion step over this recursion point. The programming of the input recursion point is set the same way. We only have to consider one special case: If the blocks B_{b_i} and B_{b_j} are neighbours (i.e. $b_j = b_i + 1$) that it is $r_1^x = r_0^x$ in this setting. So, the path beween the poles with labelling i and j enters the next recursion step graph at node r_1^x and leaves it also at r_1^x. So the programming bit of r_1^x has to be 0 in this case (and so the programming bit of r_0^x which is set first, is overwritten).

The last step is to set the input and output vectors of the two blocks. This is done in lines 26 and 27. The output vector of block B_{b_i} is set at i'-th value to the x-th recursion point and the input vector of block B_{b_j} is set at the x-th value to the j'-th pole in this block.

We repeat these steps until we achieved that for all edges $e \in E$. We note that we now have set all input and output vectors of all blocks in B. So, we can edge-embed all of them with the block edge-embedding (see Section 4.2.1) in line 30. After that we go to the next recursion steps. For all of the 4 subgraphs of G_1 in the supergraph (i.e. for all graphs in S), we call the same procedure again with $S_i \in S$ and $R_i \in R$.

4.2.4 Edge-Embedding for k-way Split EUG Constructions

In this section we give our approach for the edge-embedding task for Valiant's k-way split EUG construction [Val76; LMS16]. We only provide the main differences to the edge-embedding of the 4-way split EUG construction since they are similar.

1. k-way Block Edge-Embedding: For the k-way split EUG construction, we design a block with k input recursion points, k output recursion points and k poles. Therefore, we try to minimize the size of the block especially for PFE applications where we can compute XOR gates for free [KS08b] or a general UC construction for any application.

In this setting, we denote with $B_{k,k}^{P(x)}$ the programmable block of size x with poles $p_1, \dots, p_k$. B is topological ordered with labelling η and has the input [output] recursion points $r_0^1, \dots, r_0^k$ [$r_1^1, \dots, r_1^k$]. The vectors $in = (in_1, \dots, in_k) \in \{0, \dots, k\}^k$ and $out = (out_1, \dots, out_k) \in \{0, \dots, 2k-1\}^k$ denote the input and output vectors of B as described in Section 4.2.1. We notice that the values $k, \dots, 2k-1$ in out denote the recursion point targets $r_1^1, \dots r_1^k$. As for the 4-way split EUG construction we formalize the setting of the two vectors in (4.3). Therefore, we denote with $\mathcal{P}$ the set of all paths in B as in Section 4.2.1.

$$\begin{aligned}
\forall i \in \{1, \dots, k\} : in_i \neq 0 \rightarrow (r_0^i, p_{in_i}) \in \mathcal{P} \\
\forall i \in \{1, \dots, k\} : out_i \neq 0 \land out_i < k \rightarrow (p_i, p_{1+out_i}) \in \mathcal{P} \land \eta(p_i) < \eta(p_{1+out_i}) \\
\forall i \in \{1, \dots, k\} : out_i > k-1 \rightarrow (p_i, r_1^{i-k+1}) \in \mathcal{P} \\
\\
\forall in_i, in_j \in in : i \neq j \rightarrow in_i = 0 \lor in_i \neq in_j \\
\forall out_i, out_j \in out : i \neq j \rightarrow out_i = 0 \lor out_i \neq out_j
\end{aligned} \tag{4.3}$$

2. k-way Supergraph Construction: We denote with $G(n) \in \Gamma_2(n)$ the transformed graph of an boolean circuit C_f. To construct the supergraph construction of [KS16] we have to firstly split G into two $\Gamma_1(n)$ graphs G_1 and G_2 with 2-coloring as described in Section 4.2.2. The following steps differ from the edge-embedding of the 4-way split EUG construction. We only consider the next steps for the graph G_1. The procedure is the same for G_2.

$G_1 = (V, E) \in \Gamma_1(n)$ get merged into one $\Gamma_k(\lceil \frac{n}{k} \rceil)$ graph $G_{1_m} = (V', E')$. Therefore, we redefine the mapping θ_V that maps a node $v_i \in V$ to a node $v_j \in V'$. In this scenario, k nodes in V build one node in V', so $\theta_V(v_i) = v_{ceil \frac{i}{k}}$. The mapping of the edges θ_E is the same as in the 4-way split EUG construction (see Section 4.2.2).

The next step is that $G_{1_m} \in \Gamma_1(\lceil \frac{n}{2} \rceil)$ gets splitted into k $\Gamma_1(\lceil \frac{n}{k} \rceil)$ graphs. This can be done with a k-coloring. A directed graph $K = (V, E)$ can be k-colored, if we can create k sets $E_1, \dots, E_k$ covering the following conditions:

$$\begin{aligned}
\forall i, j \in \{1, \dots, k\} : i \neq j \rightarrow E_i \cap E_j = \emptyset \\
\forall e \in E : \exists i \in \{1, \dots, k\} : e \in E_i \\
\forall i \in \{1, \dots, k\} : \forall e = (v_1, v_2) \in E_i : \neg \exists e' = (v_3, v_4) \in E_i : v_2 = v_4 \lor v_1 = v_3
\end{aligned} \tag{4.4}$$

According to Kőnig's theorem, k-coloring of Γ_k graphs is possible with a dedicated algorithm. However, we notice that we bypass the 4-coloring problem by using two times 2-coloring. This procedure can be done if k is a power of 2.

The rest of the supergraph construction is the same as for the 4-way split EUG construction.

3. k-way Edge Embedding Algorithm: The edge-embedding algorithm for k-way split EUG constructions is the same as shown in Listing 4.1. We only have to replace every 4 with a k.

5 Implementation

[KS16] provide the first toolchain for a PFE application that uses Valiant's UC as public function. They use the the ABY framework [DSZ15] and the Fairplay compiler [MNP+04; BNP08] for the PFE application and the circuit transformation. We summarise the framework of [KS16] and detail the parts which are most important for our UC implementation in Section 5.1.

The UC module of their compiler can easily be exchanged with our UC module. We explain our UC module and the most interesting programming aspects in Section 5.2.

5.1 C++ UC Compiler

The first published implementation of a UC especially for a PFE application is provided by [KS16]. Therefore, we reuse their toolchain for our implementation. The architecture of it is visualized in Figure 5.1 [KS16].

The UC Compiler module is reimplemented, except for the first two steps due to our gained modularity in Chapter 4. We detail some aspects of its implementation in Section 5.2. However, we detail the single steps in this section, which can also be found in [KS16].

Preprocessing: Compiling Input Function f to a Circuit Description C_f: [MNP+04] provide their format *Secure Function Definition Language (SFDL)* where one can describe their functions in a high-level , C-like, language. Additionally, [MNP+04; BNP08] develop their so-called *Fairplay* compiler to translate functions f in SFDL format to a circuit description C_f by defining another format called *Secure Hardware Definition Language (SHDL)*. The Fairplay compiler is extended to *FairplayPF* in [KS08a] which ensures that the resulting circuit C_f only contains gates with indegree 2. Since Valiant's UC construction in [Val76] also requires circuits with fanin 2, [KS16] decide to use the FairplayPF extension of [KS08a]. Therefore, the result of the preprocessing step is a circuit $C_f = C_{u,v}^{k,l\geq 2}$ in SHDL format, which computes the function f.

First Step: Modifying C_f to a valid input Circuit C'_f for Valiant's UC: Valiant's UC ([Val76]) requires an input circuit with fanin and fanout 2. While the FairplayPF extension of [KS08a] ensures a fanin 2 circuit, it has no restrictions of the outdegree of gates. Therefore, [KS16] provide a function to translate the circuit C_f with fanin 2 to a circuit $C'_f = C^{k^*}_{u,v}$ with fanin and fanout 2. They use $k^* - k$ copy gates for eliminating the extra fanouts of the gates, which was shown by Valiant in [Val76]. We refer to [KS16; Val76] for more information (we provide a shortly description in Section 2.2) and use the method and implementation of [KS16] for this step. We note that $k \leq k^* \leq 2k + v$.

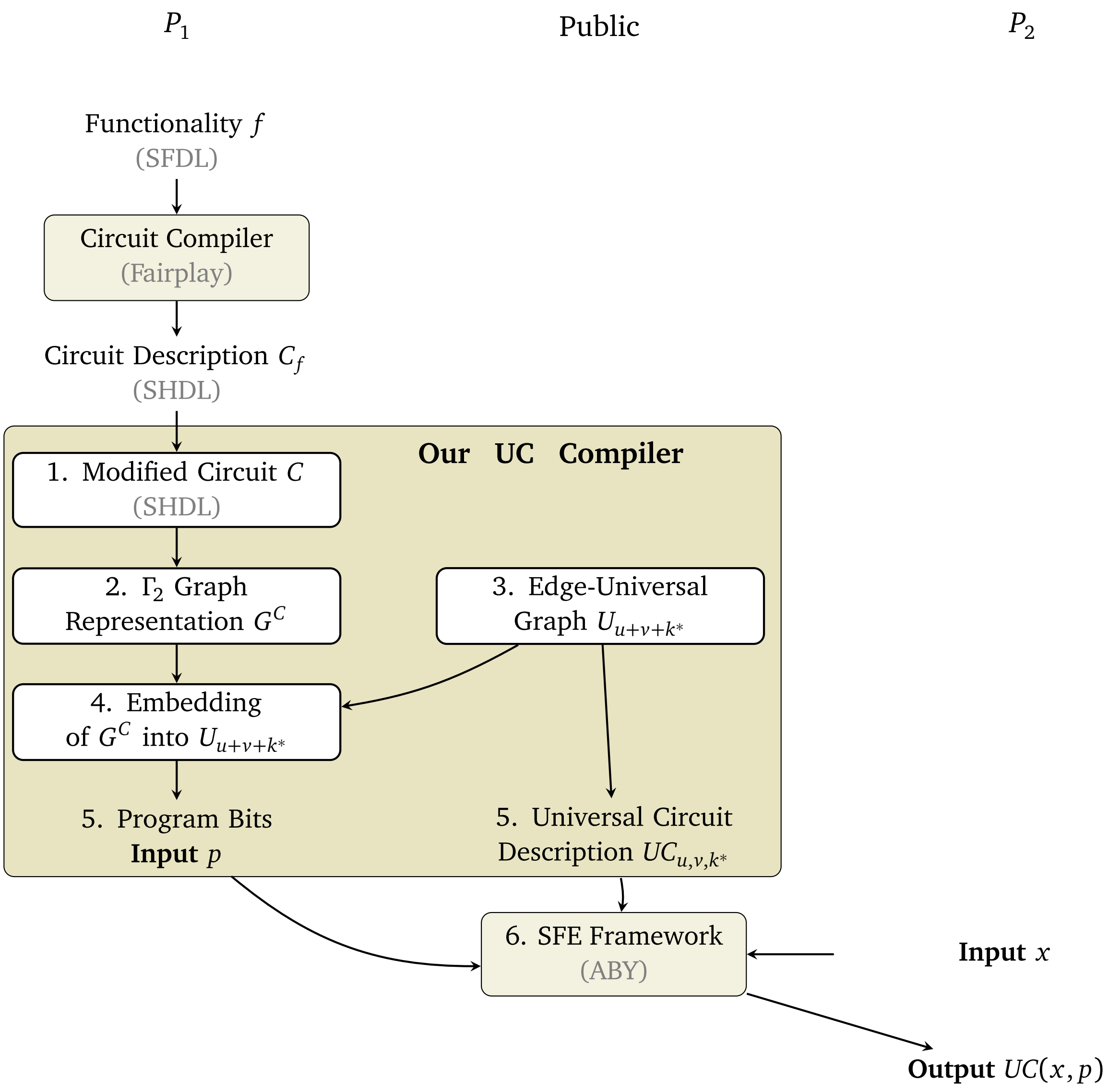

P_1
Public
P_2
Functionality f
(SFDL)
Circuit Compiler
(Fairplay)
Circuit Description C_f
(SHDL)
Our UC Compiler
1. Modified Circuit C
(SHDL)
2. Γ_2 Graph Representation G^C
3. Edge-Universal Graph U_{u+v+k^*}
4. Embedding of G^C into U_{u+v+k^*}
5. Program Bits Input p
5. Universal Circuit Description UC_{u,v,k^*}
6. SFE Framework
(ABY)
Input x
Output $UC(x,p)$

Figure 5.1: Toolchain for UC and PFE of [KS16, Figure 6]

Second Step: Transforming the Circuit C'_f to a Γ_2 Graph G: The transformation of a circuit $C'_f = C^{k^*}_{u,v}$ into a $\Gamma_2(u+v+k^*)$ graph $G = (V, E)$ is described in Section 2.2. After this transformation, we define a topological ordered labelling η for G in which every input node v_i has a labelling of $1 \leq \eta(v_i) \leq u$ and every output node v_j is labelled with $v \leq \eta(v_j) \leq (u+v+k^*)$. Graph G is the specification for the edge-embedding of the EUG U generated in the next step, i.e. G gets edge-embedded into U (see Sections 3.1.2 and 4.2).

Third Step: Creating an EUG for Γ_2 graphs $U_n(\Gamma_2)$: An EUG for Γ_2 graphs denoted by $U_n(\Gamma_2)$ can be constructed by creating two instances of $U_n(\Gamma_1)$ as mentioned in Section 3.1.2. These two instances get merged to $U_n(\Gamma_2)$ so that one instance builds the first inputs and outputs and the second instance builds the second inputs and outputs of the gates. Since we know the number of inputs, outputs and gates of C'_f and the number of nodes in $G = (V, E)$, we can set $n = |V| = u + v + k^*$.

We create the EUGs with Valiant's 4-way split EUG construction [Val76] with our optimizations in Section 4.1 and Figure 4.1. The description of the EUG $U_n(\Gamma_2)$ is public for both parties in PFE context while the programming of the nodes are private for one party.

Fourth Step: Programming $U_n(\Gamma_2)$ so that it computes f: Now that we have the EUG $U_n(\Gamma_2)$ and the description of the function f as $\Gamma_2(n)$ graph G, we can program U so that it computes f. Therefore, we edge-embed U into G which is possible for any $G \in \Gamma_2(n)$, what is proven by [Val76; KS16].

[KS16] use their supergraph construction which defines the paths between the poles uniquely for Valiant's 2-way split EUG construction [Val76]. However, since we use Valiant's 4-way split EUG construction [Val76] for our implementation, we extend their supergraph construction by providing an algorithm for this construction as well as an approach for k-way split EUG constructions [LMS16], which is detailed in Section 4.2.

The programming bits of the nodes are set during the edge-embedding process since the paths between the poles are set there.

Fifth Step: Generating the Output Circuit Description of U and its Programming Bits: The last step for the UC compiler is to generate two files, one for the circuit description of U_n and one for the programming of U_n.

Before we can create these two files, we have to topological order U_n, i.e. we define a topological ordered labelling η_U on U_n. This is important since no outputs of a node can be inputs of a node having a lower labelling value.

The file which includes the programming of U_n can be created as follows: For every node in topological order, we write in a new line the programming bit of the node. If a node is an X or a Y switching block, we only have to write the programming bit of them. However, if a node is a UG, it has 4 programming bits. In this case, we consider the 4 programming bits as binary value and translate it to a decimal value, i.e. we write $2^3c_1 + 2^2c_2 + 2c_3 + c_4$ for programming bits $c = (c_1, c_2, c_3, c_4)$. In any other case, we can skip this node since it has no programming bit, i.e. this node is either an input or output pole or a copy switching block having only one input.

Now we detail the other output file, which includes the description of the circuit U_n. [KS16] use a circuit description format which starts with enumerating the inputs and ends with enumerating the outputs. X and Y switching blocks are denoted by X and Y while UG are denoted by U. The remaining nodes are replaced by wires since they only have one input. We have to translate the node labelling to a wire labelling. Therefore, we assume that every output edge in U_n is labelled with the labelling η_U of the node. This can easily be done for Y switching blocks and UG since they both have only one output (note: two outputs of the UG have the same value). An X switching block has two outputs which can have different values, i.e. we need two wire labelling for X switching blocks. The problem is, that the labelling of the next nodes get shiftet by 1 for each X switching block with a lower labelling value. We have the same problem for nodes with only one input, since we ignore them by simply translating them into wires. Therefore, we define a new labelling for the nodes in $U_n = (V', E')$ with

$$\eta'_U(v) = \eta_u(v) + |\{w \in V' | x(w) \land \eta(v) > \eta(w)\}| - |\{w \in V' | i(w) \land \eta(v) > \eta(w)\}| \qquad (5.1)$$

where $x(v)$ $[i(v)]$ describes a predicate which is true if $v \in V'$ is an X switching block [has one input]. With this labelling, we can create the lines for the circuit description as follow:

$$\begin{array}{c} U \quad in_1 \quad in_2 \quad \eta'_U(v) \\ X \quad in_1 \quad in_2 \quad \eta'_U(v) \quad \eta'_U(v)+1 \\ Y \quad in_1 \quad in_2 \quad \eta'_U(v) \end{array} \qquad (5.2)$$

The values for in_1 and in_2 can be found at the parent nodes labelling.

Sixth Step: PFE with UC: [KS16] use PFE as application example for UC by using the *ABY framework* of [DSZ15] for secure two-party computation. ABY is compatible with the circuit format constructed in the previous step. One only has to provide an implementation of X and Y switching blocks as well as UG. [KS16] provide these implementations with considering the free XOR computing [KS08b], which is also implemented in ABY [DSZ15].

The ABY framework generates a circuit from the output circuit description file. The private function f is represented in the programming file of the previous step, denoted by p_f. A second party can provide another input x so that ABY can compute $UC(p_f, x) = f(x)$ in a privacy-preserving manner.

5.2 Universal Circuit Module

In this section, we provide some implementation details of the UC module of the toolchain from [KS16]. Therefore, we explain the single classes and their relationship.

We implement the UC module with C++ as already mentioned. It contains the classes *ValiantUC*, *ValiantEUG*, *Block* and *UCNode*. The modularity of our construction is an important factor also for the implementation. We implement every class in a way that we can exchange every component without loosing the robustness of our construction. This means, e.g., that one can implement a 3-way split EUG construction easily by only exchanging the construction of the EUG and the edge-embedding part - the rest of the code can be reused without problems. We explain the single classes in the following.

ValiantUC: This class represents the whole UC construction consisting of two instances of the class *ValiantEUG*. It starts the construction of the whole EUG for Γ_2 graphs depending on the number of nodes, which are the sum of the number of inputs, outputs and gates of the input circuit. Additionally, it also starts the edge-embedding. Therefore, the supergraph construction of [KS16], which is extended in Section 4.2.2, is offered as input. We notice that the construction of the supergraph is not included in our module - it is an external module also provided by [KS16]. After the edge-embedding is initialized, each of the two *ValiantEUG* instances get one of the first two Γ_1 graphs in [KS16] supergraph construction as input. The last task of this class is to generate the two output files (see the fifth step in Section 5.1). Therefore, it also contains the topological ordering of the nodes, which is indispensable for the generation of the output files.

ValiantEUG: The *ValiantEUG* class contains one of the two Γ_1 EUG in Valiant's UC construction [Val76]. It contains all blocks of its EUG. The two main tasks of this class are the construction of the EUG as well as the edge-embedding of the recursion points and setting the inputs of the block edge-embedding (see Section 4.2.1).

An instance of *ValiantEUG* is not a complete EUG. It contains a list of poles, a list of blocks where each block contains at least one pole as well as the recusion points. Additionally, it contains a list of its 4 children, which are new instances of *ValiantEUG* which have one of the 4 recursion point layers as poles. We choose this design since the edge-embedding task can be easily solved recursively (see Section 4.2.3).

Block: This class represents a block within an EUG (see Section 4.1). It contains a list of all its nodes. The tasks of a block are the construction of all its nodes and the edges between them as well as the block edge-embedding (see Section 4.2.1). We detail our methods for the block edge-embedding in Section 5.2.1.

UCNode: This class represents a node in Valiant's EUG construction [Val76]. It contains a list of parent and children *UCNodes*, respectively, its topological ordered value as well as its programming bits which are set during the edge-embedding. The only task of the *UCNode* class is to provide information to the other classes.

5.2.1 Methods for Block Edge-Embedding

In this section, we provide some instructions for the block edge-embedding task.

The block edge-embedding programs the nodes within the block, i.e. each block programs their nodes themselves (see Section 4.2.1). In general, the result depends on the block type (head, body, tail), the input vector in and the output vector out. We provide two methods for the block edge-embedding: using a Lookup Table (LUT) or analyzing the behaviour.

Option 1: Lookup Tables: The nodes can be programmed by LUT. Therefore, we create a table for each combination of input and output vectors and can read the programming for the individual nodes in the result columns. Since $in \in \{0, \dots, 4\}^4$ and $out \in 0, \dots, 7^4$, we have a table of size $5^4 \cdot 8^4 = 9\,834\,496$. This is a very large value and it needs a very high effort to create such a LUT. However, we can reduce the number of entries in the LUT enormously.

At first, we can remove the dummy values by replacing them with other values. Each vector consists of pairwise different values, i.e. we replace every dummy input with a value between 1 and 4 which does not occur in the input vector. For the output vector, we set the dummy values to a value between 4 and 7 since the values 1, 2 and 3 conflict with the input vector. So, we can distinguish if a target pole belongs to the input or the output vector. We can give an upper bound for this with $4! * \frac{7!}{3!} = 20\,160$. This are nearly 500 times less entries than without this optimization.

We can reduce this value again by splitting our LUT to two LUTs. The new input vector builds the first LUT and the output vector the second LUT, i.e. the second LUT can overwrite the programming of the first LUT which is necessary if there is a pole as target. With this, we have $4! = 16$ entries in the first LUT. However, the two side nodes between the poles p_{4i+2} and p_{4i+3} in Figure 4.1a decide whether the inputs or the both upper poles are forwarded to the both lower poles. This can cause a problem if one input and one of the both upper poles are forwarded to one of the both lower poles, e.g. if the input is forwarded through the right side and the output shall also be forwarded through the right side, in which case the input would get lost. We can fix this problem by adding a new input to the second LUT, denoted with z, which is set to 0 [1] if exactly one original input is forwarded to one of the lower both poles through the left [right] side. This value z can be determined easily after the first LUT result.

We notice that z has only been considered in the output vector, if exactly one of the both upper poles is forwarded to one of the both lower poles. So, we have another upper bound for the second LUT with $2 \cdot \frac{7!}{3!} = 1680$. Though this is a high value, it is only an upper bound. In reality, only the first value of the output vector has 7 options to go. If $out_1 \neq 1$, the second value out_2 has only 5 options instead of 6 since out_2 cannot be 1 (see Section 4.2.1) and $out_1 \neq out_2$. The same goes also for out_3 and out_4. We let a program determine that the second LUT has 404 entries, which are 420 entries for both LUTs. This is an enormous reduction from nearly 10 million entries to only 420 entries. However, since these are still many entries, we use the second, more efficient, option for our implementation.

Option 2: Analyzing the Nodes: The second option is to analyze the single nodes how they behave for certain input values. Therefore, we can detect multiple nodes in different blocks, which have the same programming for the same input and output vectors, e.g. the first 4 nodes of the body and tail blocks with 3 and 4 poles have the same programming depending on the input vector in Figure 4.1. This can be done for various nodes in different blocks. We firstly have only to consider the 15 nodes of the body block. Each node of the head block can be mapped to a node in the body block having the same programming for the same output vector (note: a head block has no input vector). The same goes for the tail block with 4 and 2 nodes. Only the tail blocks with 3 and 1 pole differ. Each of them contain one node (the lowest nodes) which cannot be mapped to a node in the body block. So all in all we only have to consider 17 node programmings for the different block types. We notice that we can do this optimization also for the LUT option which is mentioned above.

The nodes in blocks which build the recursion base blocks (i.e. a block which has no incoming and outgoing recursion points, since it builds the end of the recursive construction), also cannot be mapped to a node in the body block. However, these can be programmed easily.

We can also create the help input vector from the previous option, in which each dummy input of the original input vector get replaced by a value which does not occur so far in this vector. This help vector can be used for the upper permutation network of the body block (consisting of 4 nodes). The path from a recursion point to the first (second) pole has to be forwarded through the third (fourth) node in this network. So, the programming of the first two nodes only have to fulfill this task. The task of the third (fourth) node is to forward the correct recursion point to the first (second) pole. So, the programming of this permutation network can be easily done. Creating another help vector for the lower permutation network, which consists the destinations for the lower recursion points for the 4 inputs of this permutation network, makes the programming of this permutation network as easy as above.

The node between the first two poles can also easily programmed by setting the programming bit to 1 if $out_1 = 1$. Otherwise, it can be always set to 0. The same goes for the node between the third and fifth pole.

The programming of the remaining poles is more difficult since we always have to know which are the real destinations of each input of a node (here we use the original input vector and not the help input vector without dummy inputs). However, we can easily determine the destinations of the inputs of the nodes and can forward them as we need them.

This option is more error-prone due to the manual effort of optimizing the possibilities. However, we decide to use it over the first option since we predict its performance to be better than the LUT option.

5.2.2 Topological Ordering

In this section, we provide our realization of the topological ordering process for the EUG for Γ_2 graphs.

The EUG $U_n(\Gamma_2)$ has to be topological ordered in order to create the two output files of the UC (see Section 5.1, fifth step). Therefore, we use the Depth-First Search (DFS) based topological ordering algorithm, which can be found e.g. in [CLRS01, Section 22.4]. The algorithm starts at a node, marks this node, checks if all its children are marked to set a topological value to this node, otherwise it marks all the children and the same process is performed for their children. So, this process is done recursively.

Since we have an EUG in which we can create a path from the first pole to any other node or pole, we can start the algorithm at the first pole and all nodes get a topological value. However, since the real input [output] poles do not have inputs [outputs], we have to delete the incoming [outgoing] edges from these poles. Therefore, the algorithm labels not every node (especially the real input poles) with a topological value. We can fix this by labelling the real input poles by hand in ascending order (starting with 1) and run the algorithm twice starting with both children of the first pole.

The first thing we have to do is to determine the size of the EUG since the algorithm starts with the biggest value for the topological ordering and ends with the value 1 at the first pole. This can be done by counting the nodes and poles of each block in the EUG.

The algorithm is recursive and this becomes a problem with large size EUG. The number of recursion pointers on the stack depends on the depth of the EUG. For a high depth, we get problems with the stack due to the high number of entries so that the stack is full - the program crashes. Therefore, we use the *stack* class from C++ where we can manage a stack by hand. So, we push every node which is visited on the stack and remove them when they get their topological value. With this method we get much less entries on the step since we do not have to save the temporary values for each node too. We show our C++ procedure for the topological ordering in Listing 5.1.

Listing 5.1: Topological Ordering Algorithm in C++

```cpp
void ValiantUC::topologicalOrdering (uint32_t inputs) {
  uint32_t topValue = this->size - 1;
  std::stack<UCNode*> dfs;
  for (int i = 0; i < inputs; i++) {
    this->poles[i]->setTopologicalVisited (true); // mark node
    this->poles[i]->setTopologicalNumber(i);
    this->topOrderedNodes[i] = this->poles[i];
  }
  UCNode *currentNode;
  // push both children of the first pole on the stack
  dfs.push(this->poles[0]->getChildren()[0]);
  dfs.push(this->poles[0]->getChildren()[1]);
  while (!dfs.empty()) {
    currentNode = dfs.top();  // top node on the stack
    currentNode->setTopologicalVisited(true);
    bool foundSomething = false;
    for (auto child : currentNode->getChildren()) {
      // check if all children are visited
      if (!child->getTopologicalVisited()) {
        dfs.push(child);
        foundSomething = true;
        break;
      }
    }
    if (foundSomething) {
      continue;
    }
    currentNode->setTopologicalNumber (topValue);
    topOrderedNodes[topValue] = currentNode;
    topValue--;
    dfs.pop();
  }
}
```

6 Evaluation

In this chapter, we compare the sizes of the 2-way split and the 4-way split UC constructions. Therefore, we take a look at the sizes of the EUG constructions in Section 6.1. Thereafter, we compare the number of AND gates of the whole UC construction in Section 6.2. We finish this chapter with providing a future work direction in Section 6.3.

6.1 Size of k-way EUG

Considering Valiant's 4-way split EUG construction as chain of blocks in which each chain can have 4 sub-chains simplifies the programming of the nodes. The underlying idea comes from [LMS16], who regard a group of k poles as block (see Section 3.2.2). Nearly every block consist of k incoming and k outgoing recursion points (only the head and the tail of the chains behave differently). In this section, we compare the sizes of Valiant's 2-way and 4-way EUG construction and develop a formula for calculating the size of a k-way split EUG construction provided that the sizes of the single blocks are known.

We denote with $b_i(k)$ and $z_i(k)$ the number of blocks and the number of recursion points at the i—th recursion step of Valiant's k-way split EUG construction, respectively. This means, the base chain consists of $b_0(k) = \lceil \frac{n}{k} \rceil$ blocks and has $z_0(k) = \lceil \frac{n}{k} \rceil - 1 = b_0(k) - 1$ recursion points when having n poles. It is easy to see that the number of poles at the i—th recursion step is $z_{i-1}(k)$. Therefore, we can define $b_i(k)$ and $z_i(k)$ as follow:

$$b_i(k) = \begin{cases} \lceil \frac{n}{k} \rceil & \text{if } i = 0 \\ 0 & \text{if } z_{i-1}(k) = 0 \\ \lceil \frac{z_{i-1}(k)}{k} \rceil & \text{else} \end{cases} \tag{6.1}$$

$$z_i(k) = \begin{cases} b_0(k) - 1 & \text{if } i = 0 \\ 0 & \text{if } b_i(k) = 0 \\ b_i(k) - 1 & \text{else} \end{cases} \tag{6.2}$$

This recursion terminates at $z_t(k) = 0$. With this approach we can calculate the size of the EUG constructions.

We now have to define a function f_k which returns the size of any block in Valiant's k-way split EUG construction. The values $b_i(k)$ and $z_{i-1}(k)$ can be used to determine such a function. We notice that we have to set $z_{-1}(k) = n$ for the case $i = 0$. The last input of the function f_k is a value $1 \leq j \leq b_i$ which can be interpreted as the j-th block in the current chain of blocks. Now that we have this, we can define $f_k(j, b_i, z_{i-1})$ to return the size of the j-th block in the i-th recursion step chain. The formula for $k = 4$ is given in Equation (6.3).

$$f_4(j, b_i, z_{i-1}) = \begin{cases} 7 & \text{if } b_i = 1 \wedge z_{i-1} = 4 \text{ (Recursion Base Block (4 poles))} \\ z_{i-1} & \text{if } b_i = 1 \wedge z_{i-1} \neq 4 \text{ (Recursion Base Block (}z_{i-1}\text{ poles))} \\ 14 & \text{if } b_i > 1 \wedge j = 1 \text{ (Head Block)} \\ 14 & \text{if } b_i > 1 \wedge j = b_i \wedge z_{i-1} \mod 4 = 0 \text{ (Tail Block (4 poles))} \\ 11 & \text{if } b_i > 1 \wedge j = b_i \wedge z_{i-1} \mod 4 = 3 \text{ (Tail Block (3 poles))} \\ 7 & \text{if } b_i > 1 \wedge j = b_i \wedge z_{i-1} \mod 4 = 2 \text{ (Tail Block (2 poles))} \\ 4 & \text{if } b_i > 1 \wedge j = b_i \wedge z_{i-1} \mod 4 = 1 \text{ (Tail Block (1 pole))} \\ 19 & \text{else (Body Block)} \end{cases} \tag{6.3}$$

Since we compare the 4-way split EUG construction with the 2-way split EUG construction, we also provide a definition of $f_2(j, b_i, z_{i-1})$ in Equation (6.4).

$$f_2(j, b_i, z_{i-1}) = \begin{cases} 7 & \text{if } b_i = 1 \wedge z_{i-1} = 4 \text{ (Recursion Base Block (4 poles))} \\ z_{i-1} & \text{if } b_i = 1 \wedge z_{i-1} \neq 4 \text{ (Recursion Base Block (}z_{i-1}\text{ poles))} \\ 4 & \text{if } b_i > 1 \wedge j = 1 \text{ (Head Block)} \\ 4 & \text{if } b_i > 1 \wedge j = b_i \wedge z_{i-1} \mod 2 = 2 \text{ (Tail Block (2 poles))} \\ 2 & \text{if } b_i > 1 \wedge j = b_i \wedge z_{i-1} \mod 2 = 1 \text{ (Tail Block (1 pole))} \\ 5 & \text{else (Body Block)} \end{cases} \tag{6.4}$$

Having $f_k(j, b_i, z_{i-1})$, we can provide a general formula for the size of a k-way split EUG construction in Equation (6.5).

$$size_{\text{EUG}}(k) = \sum_{\substack{i=0 \\ b_i \neq 0}}^{t} k^i \sum_{j=1}^{b_i} f_k(j, b_i, z_{i-1}) \tag{6.5}$$

Since $b_i = 0$ for $i > t$, the sum terminates in any case. We compare the 2-way and 4-way split EUG constructions with this formula. The results are shown in Figure 6.1.

We see that up to $n = 40\,000$ there is only an insignificant difference between the 2-way and 4-way split EUG constructions. The 4-way split EUG constructions has smaller sizes for $n > 40\,000$ (this point is marked with a green line in Figure 6.1). There is an exception in the area of $90\,000 < n < 130\,000$ in which the 2-way split EUG constructions has a better size in some cases (this area is marked with red lines in Figure 6.1). However, from a certain value n, the 4-way split EUG construction always yields better results since there are linear terms which dominate the logarithmic function term when the logarithmic function is not yet big enough.

The reason for the fact that in some areas the 2-way split performs better are the jumps, e.g. at $n = 130\,000$ for the 2-way split and at $n = 85\,000$ for the 4-way split. These jumps are the results when the number of recursion for the EUG constructions increases by 1. Lets assume we have the maximum number of poles n where we have i recursions and the i-th recursion chain consists of one block with 4 poles and if we increase n by 1, there occurs a fifth pole

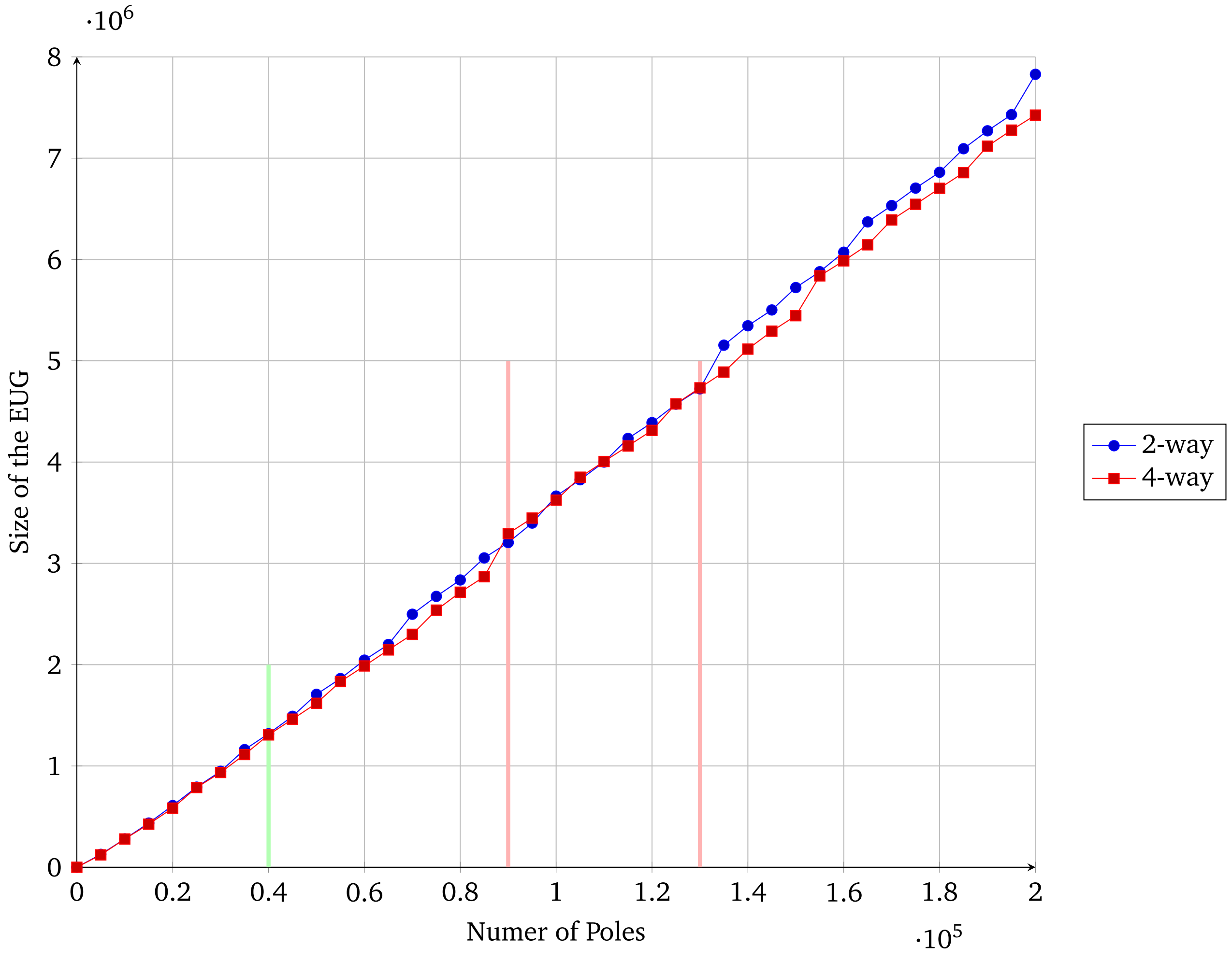


Figure 6.1: Comparison between the 2-way and 4-way split EUG constructions with up to 200 000 poles

in the i-th recursion step. The EUG construction for n poles has $4^i \cdot 7$ nodes only in the i-th recursion steps (7 is the size of the 4 poles recursion block). If we construct an EUG for $n+1$ poles, the i-th recursion step contains of 5 poles, i.e. the chain contains of two blocks - one head block of size 14 and one tail block of size 4. So, the i-th recursion steps contain now in sum $4^i \cdot 18$ nodes which is a giant jump for large i. If we compare an EUG having a tail block with 3 poles in the main chain (i.e. the 0-th recursion step), then the one larger EUG construction only exchanges this tail block of 3 poles (size 11) with a tail block of 4 poles (size 14). So, in this case the size only grows by 3. The same applies for the 2-way split EUG construction.

If the 4-way split EUG construction has such a scenario, the size grows significantly for big n - it can even get larger than the 2-way split EUG construction. However, after such a jump the 4-way split EUG construction has a large break before another big jump occurs. Therefore, the construction has much time to recover so that the 2-way split EUG construction grows over it. So, our construction has a better size for most n values, and for big enough n it always outperforms the 2-way split construction.

6.2 Size of the UC

In this section we provide the size of our UC construction.

The size for a UC is counted with the number of *AND* and *XOR* gates. An X switching block can be implemented with 1 AND and 3 XOR gates, a Y switching block with 1 AND and 2 XOR gates and a UG with 3 AND and 6 XOR gates [KS16].

At first, we calculate our theoretical size of the k-way split UC with n poles using the formula in Equation (6.5) from the previous section. However, we cannot distinguish between Y and X switching blocks that way. Therefore, we only provide an upper bound for the number of AND and XOR gates by only considering X switching blocks. The upper bound is given in Equation (6.6). We notice that a UC consists of two EUG that only share the same poles (see Section 3.2). Therefore, the size of the EUG has to be multiplied by 2. Since the number of AND and XOR gates are different for UG and the X switching blocks, we have to count them seperately. So, the first part of the equations in Equation (6.6) calculates the number of AND and XOR gates for the X-switching blocks, while the second part of the equations counts the number of AND and XOR gates for the UGs.

$$\begin{aligned} size_{\text{AND}}(k,n) &= (2 \cdot size_{\text{EUG}}(k) - 2 \cdot n) + 3 \cdot n = 2 \cdot size_{\text{EUG}}(k) + n \\ size_{\text{XOR}}(k,n) &= (2 \cdot size_{\text{EUG}}(k) - 2 \cdot n) \cdot 3 + 6 \cdot n = 6 \cdot size_{\text{EUG}}(k) \end{aligned} \tag{6.6}$$

We provide our upper bound for number of poles up to 200 000 in Figure 6.2.

As for the size of the EUG, the number of AND and XOR gates is for most n less for the UC constructed with two 4-way split EUGs. However, we see that the difference between the number of AND gates is very small even for large n, i.e. for a setting of PFE with free XOR gates evaluation [KS08b] (e.g. with ABY [DSZ15]) the difference between the two constructions is not significant for smaller n values.

6.2.1 Improvement for the UC Size

[KS16] use an improvement for their UC construction that decreases the UC size rapidly. This optimization outperforms our 4-way split construction described in Chapter 4 for n up to at least 200 000. However, we can also adapt this optimization for our UC construction.

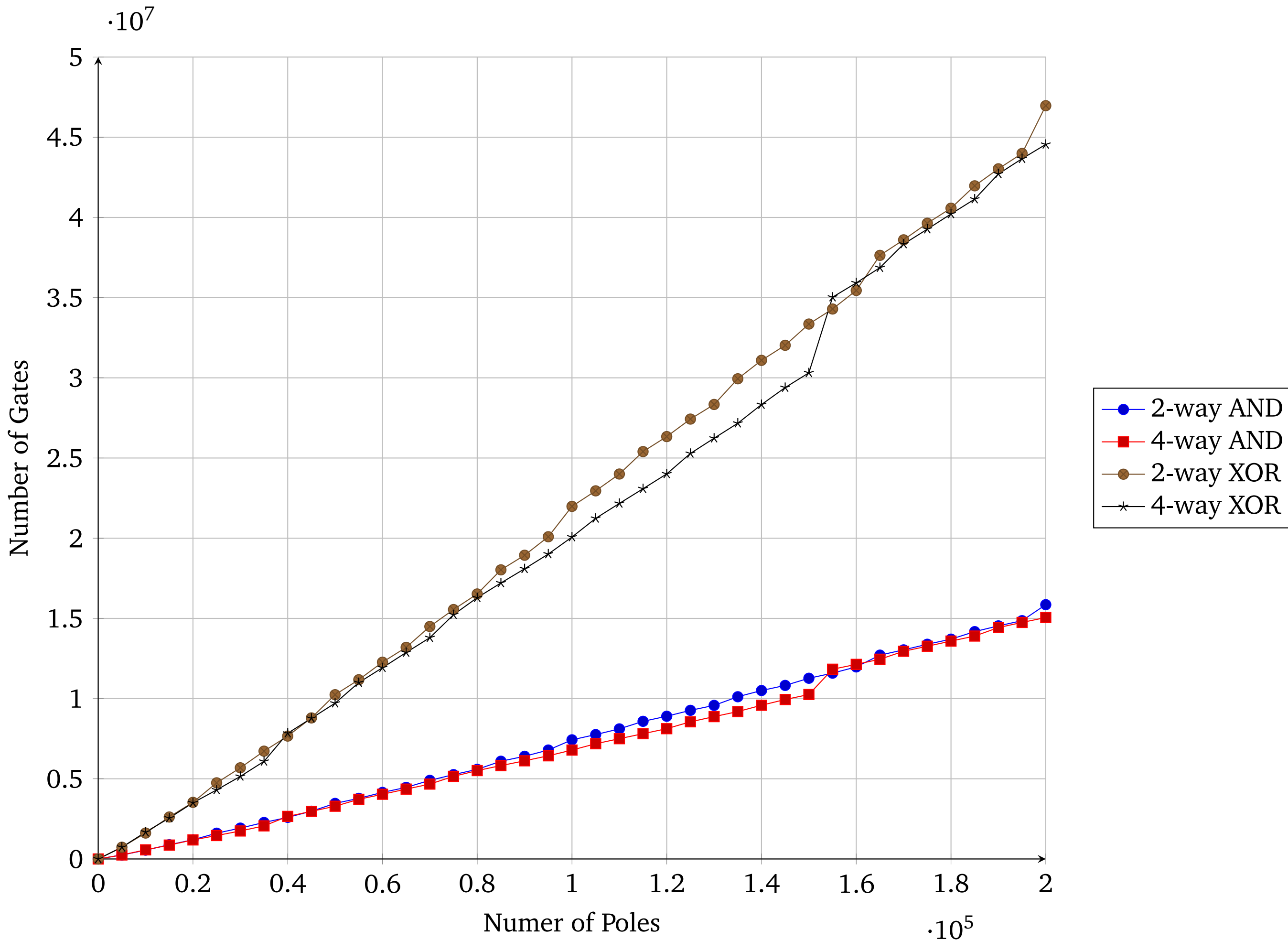


Figure 6.2: Comparison number of AND and XOR gates between UC constructed with 2-way and 4-way split EUG construction with up to 200 000 poles

Circuit	u	k	v	$k^* - k$	[KS16]	**4-way split**
AES-non-exp	256	31 924	128	14 539	$2.875 \cdot 10^6$	$\mathbf{2,866 \cdot 10^6}$
AES-exp	1 536	25 765	128	11 089	$\mathbf{2.303 \cdot 10^6}$	$2.333 \cdot 10^6$
DES-non-exp	128	19 464	64	12 290	$1.876 \cdot 10^6$	$\mathbf{1,860 \cdot 10^6}$
DES-exp	832	19 526	64	11 785	$\mathbf{1.876 \cdot 10^6}$	$1,883 \cdot 10^6$
md5	512	43 234	128	22 623	$4.250 \cdot 10^6$	$\mathbf{4,124 \cdot 10^6}$
sha-1	512	61 466	160	33 132	$6.334 \cdot 10^6$	$\mathbf{6,213 \cdot 10^6}$
sha-256	512	132 654	256	67 584	$1.448 \cdot 10^7$	$\mathbf{1.400 \cdot 10^7}$
add_32	64	187	33	58	**8 878**	9 516
add_64	128	379	65	102	**20 682**	21 984
comp_32	64	150	1	1	**4 846**	5 478
mult_32x32	64	6 995	64	5 079	$\mathbf{6.304 \cdot 10^5}$	$6,340 \cdot 10^5$
Branching_18	72	121	4	3	**4 328**	4 850
CreditChecking	25	50	1	6	**1 264**	1 477
MobileCode	80	64	16	0	**3 132**	3 410

Table 6.1: Comparison of the AND sizes between the UC implementation of [KS16] and ours (the improved one) using circuits provided by [TS15]. We emphasize the best values with bold numbers. The most part of this table are from [KS16, Table 3]. $k^* - k$ denotes the additional number of gates (see Step 1 in Section 5.1).

The optimization occurs at the tail block and the body block before the tail block. If the tail block does not contain 4 poles, the number of input recursion points can be reduced to the number of poles. So, the last two blocks of the chain can be modified so that the lowest recursion points can be reduced to the number of poles within the tail block. This reduces the size of the next recursion steps, too.

In the context of this work, we only implement the construction of this modified UC construction so that we can compare the AND sizes with the implementation of [KS16]. The edge-embedding for this improvement is not implemented so far and is left as future work.

We now look at exact AND sizes for some example input graphs provided by [TS15]. Therefore, we reuse parts of [KS16, Table 3] and add our results in Table 6.1.

We see that our UC implementation provides better results for large circuits while the implementation of [KS16] is better for small circuits. This result is expected since Valiant's UC construction [Val76] is asymptotically optimal, i.e. the AND size of our UC implementation is better for a circuit size greater than around 40 000, which also fits with our theoretical AND sizes in Figure 6.1. Therefore, one should use our improved UC construction for large circuits.

6.3 Future Work

We provide an implementation of Valiant's 4-way split UC construction. However, while our implementation fits a modular design for UC constructions, the size of the implementation of [KS16] is better. We adapt their improvements to our UC construction and recognize that this modified UC construction has better results. The implementation of the edge-embedding is missing for this optimized construction. This task is not so difficult considering our edge-embedding approach described in Section 4.2. One can modify our tail block constructions and include new block which have 4 input recursion points but only 3, 2 and 1 output recursion points.

[LMS16] calculate the number of k which minimizes the size of their k-way split EUG construction. They calculated the value $k \approx 3.147$ which is closer to 3 than to 4. Although their understanding of k-way split EUG constructions differs, their block sizes provide an upper bound for efficient k-way split EUG constructions. So, we think it could be more efficient to create a 3-way split EUG construction. Therefore, an interesting future research direction is to optimize head, tail and body blocks by hand and compare the resulting performance with the existing constructions.

List of Figures

List of Tables

Abbreviations

DFS	Depth-First Search
EUG	Edge-universal Graph
LUT	Lookup Table
OT	Oblivious Transfer
PFE	Private Function Evaluation
SFE	Secure Function Evaluation
SFDL	Secure Function Definition Language
SHDL	Secure Hardware Definition Language
UC	Universal Circuit
UG	Universal Gate

Bibliography

[ALSZ13] G. Asharov, Y. Lindell, T. Schneider, M. Zohner. **"More efficient oblivious transfer and extensions for faster secure computation"**. In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 2013, pp. 535–548 (cit. on p. 7).

[Bea91] D. Beaver. **"Efficient Multiparty Protocols Using Circuit Randomization"**. In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*. 1991, pp. 420–432 (cit. on p. 8).

[BNP08] A. Ben-David, N. Nisan, B. Pinkas. **"FairplayMP: a system for secure multi-party computation"**. In: *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*. 2008, pp. 257–266 (cit. on p. 28).

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. 2nd. Cambridge, MA, USA: MIT Press, 2001 (cit. on p. 35).

[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 pp. 28, 31, 41).

[FAZ05] K. B. Frikken, M. J. Atallah, C. Zhang. **"Privacy-preserving credit checking"**. In: *Proceedings 6th ACM Conference on Electronic Commerce (EC-2005), Vancouver, BC, Canada, June 5-8, 2005*. 2005, pp. 147–154 (cit. on p. 1).

[FVK+15] B. A. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, S. M. Bellovin. **"Malicious-Client Security in Blind Seer: A Scalable Private DBMS"**. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 395–410 (cit. on p. 2).

[GGPR13] R. Gennaro, C. Gentry, B. Parno, M. Raykova. **"Quadratic Span Programs and Succinct NIZKs without PCPs"**. In: *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*. 2013, pp. 626–645 (cit. on p. 2).

[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 nineteenth annual ACM symposium on Theory of computing*. ACM. 1987, pp. 218–229 (cit. on pp. 1, 6–9).

[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. 7).

[KOS15] M. Keller, E. Orsini, P. Scholl. **"Actively Secure OT Extension with Optimal Overhead"**. In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. 2015, pp. 724–741 (cit. on p. 7).

[KS08a] 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. 2, 7, 9, 14, 16 sq., 28 sq.).

[KS08b] 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 pp. 6 sqq., 16, 26, 31, 41).

[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.* Full version: Eprint 2016/093 at `eprint.iacr.org/2016/093.pdf`. 2016, pp. 699–728 (cit. on pp. 1 sqq., 5 sqq., 13 sq., 17 sq., 20–24, 27–32, 40–43).

[LMS16] H. Lipmaa, P. Mohassel, S. S. Sadeghian. **"Valiant's Universal Circuit: Improvements, Implementation, and Applications."** In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 17 (cit. on pp. 1 sq., 9, 13 sq., 16, 18, 26, 30, 37, 43).

[LP09] L. Lovász, M. D. Plummer. *Matching theory*. Vol. 367. American Mathematical Soc., 2009 (cit. on p. 22).

[MNP+04] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella. **"Fairplay-Secure Two-Party Computation System."** In: *USENIX Security Symposium*. Vol. 4. San Diego, CA, USA. 2004, pp. 287–302 (cit. on p. 28).

[MS13] P. MOHASSEL, S. S. SADEGHIAN. **"How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation"**. In: *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*. 2013, pp. 557–574 (cit. on p. 2).

[MSS14] P. MOHASSEL, S. S. SADEGHIAN, N. P. SMART. **"Actively Secure Private Function Evaluation"**. In: *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*. 2014, pp. 486–505 (cit. on p. 2).

[PKV+14] V. PAPPAS, F. KRELL, B. VO, V. KOLESNIKOV, T. MALKIN, S. G. CHOI, W. GEORGE, A. D. KEROMYTIS, S. BELLOVIN. **"Blind Seer: A Scalable Private DBMS"**. In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. 2014, pp. 359–374 (cit. on p. 2).

[Sch08] T. SCHNEIDER. **"Practical Secure Function Evaluation"**. In: *Informatiktage 2008. Fachwissenschaftlicher Informatik-Kongress, 14. und 15. März 2008, B-IT Bonn-Aachen International Center for Information Technology in Bonn*. 2008, pp. 37–40 (cit. on p. 4).

[Sha49] C. E. SHANNON. **"The synthesis of two-terminal switching circuits"**. In: *Bell Systems Technical Journal* 28 (1949), pp. 59–98 (cit. on p. 4).

[TS15] S. TILLICH, N. SMART. *Circuits of basic functions suitable for MPC and FHE*. 2015 (cit. on p. 42).

[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*. Ed. by Ashok K. Chandra, Detlef Wotschke, Emily P. Friedman, and Michael A. Harrison. ACM, 1976, pp. 196–203. URL: `http://doi.acm.org/10.1145/800113.803649` (cit. on pp. 1 sq., 4 sq., 9 sqq., 14 sq., 17–20, 22–26, 28 sqq., 32 sq., 42).

[Weg87] I. WEGENER. *The complexity of Boolean functions*. Wiley-Teubner, 1987 (cit. on pp. 1, 14).

[Yao82] 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 pp. 1, 6 sq.).

[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*. IEEE Computer Society, 1986, pp. 162–167 (cit. on pp. 1, 6 sq., 9).