TECHNISCHE
UNIVERSITÄT
DARMSTADT

Master Thesis

Improving Scalability of Universal Circuits for Large-Scale Private Function Evaluation

Masaud Y Alhassan

April 3, 2018

CRISP
Center for Research
in Security and Privacy

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

Supervisors: Prof. Dr.-Ing. Thomas Schneider
M.Sc. Ágnes Kiss

Thesis Statement
pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Masaud Y Alhassan, have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Darmstadt, April 3, 2018

Masaud Y Alhassan

Abstract

Private Function Evaluation (PFE) allows two parties to compute a function $f(x)$ based on their private inputs, where one party provides the function f and the other party provides the input x without either party knowing about the other's input, therefore preserving privacy. Secure Function Evaluation allows two parties to compute a global function $f(x, y)$ based on their private inputs x and y. A Universal Circuit (UC) is a globally known structure that can simulate functions. We can therefore achieve PFE using SFE by making the UC global and providing it with private inputs f and x from the two parties. The UC is a structure made up of nodes which have to be programmed (setting control bits for each node) according to the function it has to simulate.

Valiant proposed the first Universal Circuit (UC) constructions, which are recursive and use either 2-way or 4-way recursion split methods (STOC'76). The most recent UC implementation based on Valiant's 4-way UC construction was realized by Günther et al (ASIACRYPT'17), which is also modular. Lipmaa et al. generalized a k-way UC construction which is depicted in a modular manner (Eprint 2016/017). Günther et al. prove that their modular 4-way split UC construction outperforms the 2-way split UC construction by Kiss and Schneider, that is, results in asymptotically smaller UCs. Though being modular, this UC construction stores every internal state of the algorithm in memory and therefore, memory becomes a bottleneck. For this reason, this construction cannot be used in practice for large-scale UC applications.

We introduce a scalable approach of UC generation based on Günther et al.'s modular 4-way split UC generation. Our approach leverages the file system storage with very little use of memory and can yield UC sizes in the orders of billion. Also, multiple instances of our UC implementation can be run in parallel on big machines, which was not possible or reasonable to do in older UC implementations. We show our scalable file-based UC generation which makes it extremely intuitive to perform the next step: the programming of the UC. With this scalability, we can realize large-scale PFE applications, for instance remote diagnostics of smart cars for

autonomous driving, remote diagnostics of cars for automobile insurance, home automation and remote patient monitoring. Users of the aforementioned applications will no longer be concerned with sharing sensitive data to owners of the applications.

Contents

1 Introduction

Secure Function Evaluation (SFE) allows two parties to securely evaluate a function on their respective private inputs, without revealing anything other than the result to the other party. Being able to keep the respective inputs of both parties x and y private with SFE, the problem of evaluating function f privately is transformed into the problem of finding a globally known structure that can simulate functions based on some data that is provided by one party, and the input that is provided by the other party, in other words representing the function to be evaluated as input data. A Universal Circuit (UC) is a global structure for simulating any Boolean function $f(x)$. By the application of the function input bits $x = \{x_1, ..., x_u\}$, a UC can be programmed with program bits $p = \{p_1, ..., p_m\}$ to compute different functions, such that, $UC(x, p) = f(x)$.

Valiant proposed two theoretical ways of UC construction based on two recursive methods: 2-way split and 4-way split, where the recursive structure consists of either 2 or 4 sub-structures respectively [Val76]. For an input Boolean circuit of size n, Valiant's construction has asymptotically optimal size $O(n \log n)$ [Val76; Weg87]. Valiant's 2-way split UC construction method was brought to practice in two concurrent and independent works of [KS16] and [LMS16]. Lipmaa et al. proposed a generalization of Valiant's construction known as k-way split UC construction in [LMS16]. Günther, Kiss and Schneider brought into practice, Valiant's 4-way UC construction with a concrete size of $4.75n \log_2 n$ in [GKS17]. They also proposed a generic hybrid UC construction which yielded better UC sizes than the 2-way and 4-way UC constructions for all circuits tested in their evaluation.

1.1 PFE Applications of Universal Circuits

We mention some PFE applications of UCs in this section with referral to [KS16] for more details.

Private Function Evaluation with UC allows for easy integration into existing SFE frameworks. For instance, using outsource SFE to outsource UC-based PFE [KR11]. [BPSW07] shows privacy-preserving evaluation of diagnostic programs, where the owner of the program does not want to reveal the diagnostic method and the user does not want to reveal his data. Medical systems [BFK+09] and remote software fault diagnosis are both examples of such programs, where in both cases privacy-preserving is desired for the function and the user's input. Remote streaming data can be obliviously filtered using secret keywords [OI05], from

which data privacy can be preserved by using PFE to search the matching data with a private search function.

1.2 Contributions

We provide the following contributions:

Scalable Universal Circuit Generation: We design and implement a per-block 4-way UC generation based on the modular 4-way UC construction of [GKS17], which we detail in Chapters 4 and 5. Implementation presented in [GKS17] holds the UC structure in memory which poses an obvious bound on the maximum size of UC that can be generated. We show that our per-block UC generation requires memory that is only linear in the size of the input circuit n (as opposed to $O(n \log n)$ for [GKS17]) and runs efficiently using far less memory than the implementation of [GKS17] in Section 6.2. Our per-block UC generation can generate UCs of billions of nodes, making it possible to realize large-scale PFE applications for instance remote diagnostics of smart cars for autonomous driving and home automation, while preserving the privacy of the users of the application. Multiple instances of our per-block UC generation can be run in parallel on big machines. We report slower runtimes of our per-block UC generation with an average factor of 3.0 compared to the generation by Günther et al., which is acceptable because UC generation is a precomputation step in most applications and therefore, a constant runtime factor increase is not going to become a bottleneck.

Scalable Implementation Using The File System: We leverage the file system for storage in our UC generation which amounts to many files used, which we do to avoid memory overflow. We detail our algorithm in Section 5.1.3. We show that the extra disk space used in the UC generation (apart from generated UC files) is very small though the extra space can be larger in intermediate steps of the algorithm but never exceeds the maximum disk space necessary when the UC files are also created (in Section 6.2).

Scalable UC Programming: We propose steps to how the generated UC can be programmed using the supergraph construction of [KS16] and our scalable files generation, which we leave as future work. We describe the algorithm and show how our scalable generation of files makes it very intuitive to program the generated UC (Section 5.2).

1.3 Outline

In Chapter 2, we provide basic concepts in Boolean Circuits in Section 2.1, some graph theory about directed acyclic graphs in Section 2.2, a primer on SFE in Section 2.3 and some basic concepts about file systems in Section 2.4.

We discuss some existing UC constructions in Chapter 3. We begin by briefly introducing PFE in Section 3.1. We then discuss using UCs for representing functions as data in Section 3.2. Thereafter, we discuss the steps in Valiant's UC construction [Val76] and also discuss other UC constructions [KS08a; LMS16; KS16; GKS17] in Section 3.3. Next, we discuss some limitations of existing UC implementations in Section 3.4.

We detail our per-block UC generation design in Chapter 4. In Section 4.1, we discuss the modularity of Günther et al.'s 4-way UC generation. Thereafter, we detail our per-block UC generation in Section 4.2. We then discuss edge-embedding for Valiant's 4-way UC construction [Val76] and our per-subgraph edge-embedding in Section 4.3.

In Chapter 5, we detail the implementation of our per-block UC generation in Section 5.1. We then discuss how our scalable UC and files generation can be leveraged to program the UC in Section 5.2.

We present the results of our experiments in Chapter 6. We begin by discussing the evaluation criteria used in our experiments in Section 6.1. Thereafter, we discuss our evaluation results, comparing our benchmarks with the 4-way UC construction of [GKS17] using measurements from the maximum memory consumption, UC generation runtime and maximum disk space usage in Section 6.2.

Finally in Chapter 7, we conclude our work and discuss future work in Sections 7.1 and 7.2, respectively.

2 Preliminaries

In this chapter, the necessary background to Valiant's Universal Circuit construction is provided. We begin by discussing gates and Boolean circuits in Section 2.1. Then we move on to the underlying graph theory upon which Valiant built his UC constructions in Section 2.2. Next is a primer on file systems, then finally we describe Valiant's UC constructions.

2.1 Boolean Circuits

In this section, we begin by discussing logic gates in Section 2.1.1 which are the precursor to Boolean circuits. We then discuss Boolean circuits in Section 2.1.2.

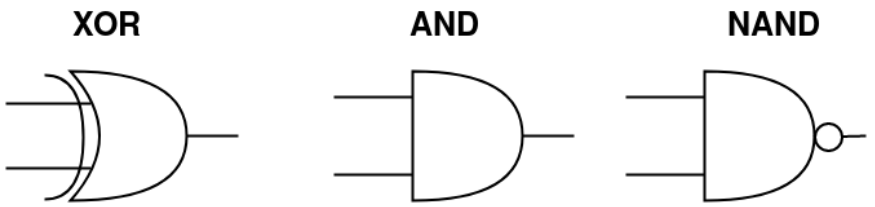
2.1.1 Logic Gates

Gates, also known as logical gates, are used for binary computations. A gate implements a Boolean function $f : \{0,1\}^m \rightarrow \{0,1\}^n$ that maps m inputs to n outputs. That is, they take as input m binary values and return as output n binary values. The most common gates include AND, OR, XOR, NOT, NAND, for which $m \geq 1$ and $n = 1$.

As an example, let us consider the 2-input, 1-output AND gate. For an AND gate with two inputs, a binary output of 1 is only obtained when both inputs are 1. Additionally, the AND gate gives a 0 output if either (or all) of its inputs is (are) 0. The table that describes the configuration and functionality of a given gate is known as a gate table. The gate tables of a few logic gates are shown in Figure 2.1

2.1.2 Boolean Circuits

Logical gates are a precursor to Boolean circuits. Boolean circuits are used in engineering to represent and model logical functions used in components such as registers in computers. A Boolean circuit is implemented with logical gates and is defined by $C^k_{u,v}$ for u inputs, v outputs and k logical gates in the given Boolean circuit. The inputs and outputs u and v denote input and output wires of the Boolean circuit. A Boolean circuit can have several inputs as well as several outputs. The output wire of a logical gate in a Boolean circuit can



x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x NAND y
0	0	1
0	1	1
1	0	1
1	1	0

Figure 2.1: Gate tables of selected example gates

serve as the input wire of another logical gate in the same Boolean circuit. An example is shown in Figure 2.2a.

2.2 Graph Theory

A Boolean circuit with u inputs, v outputs and k gates denoted by $C^k_{u,v}$ can be represented as a Directed Acyclic Graph (DAG). The Boolean circuit $C^k_{u,v}$ can therefore be transformed into a DAG by creating a node for each input, gate and output. For each wire in the circuit, we introduce a directed edge. This is called an acyclic graph because during traversal of the graph, each node is visited only once and this is done in a sequence known as topological order. We wish to define an ordering for the DAG in $\Gamma_d(n)$ (where n is the size of the DAG and d is the maximum number of incoming or outgoing edges of nodes in the DAG) such that, all nodes $V = \{a_1, ..., a_n\}$ are mapped as $\eta : V \rightarrow \{1, ...n\}$. This mapping is called a topological order if $(a_i, a_j) \in E \Rightarrow \eta(a_i) < \eta(a_j)$ and $\forall a_1, a_2 \in V : \eta(a_1) = \eta(a_2) \Rightarrow a_1 = a_2$. To provide a clearer description, a topological ordering of a DAG with nodes $V = \{a_1, a_2, ..., a_n\}$, is a linear ordering of the nodes such that for every $E_{i,j}$ belonging to $E \subset VxV$ from a_i to a_j, a_i is ordered before a_j. The nodes are therefore traversed in this defined order. Obtaining a topological order comes with a complexity of $O(n + dn)$ [KS16]. A given DAG can have more than one topological order, since the starting node as well as the strategy on finding the next node can differ significantly from each other. For example, two possible topological orders of the DAG in Figure 2.2b are $\{a_2, a_1, a_3, a_4, a_5\}$ and $\{a_1, a_2, a_3, a_4, a_5\}$. Topological ordering is further discussed in Chapter 4. Considering a DAG with nodes $V = \{a_1, ..., a_n\}$, if node a_1 is

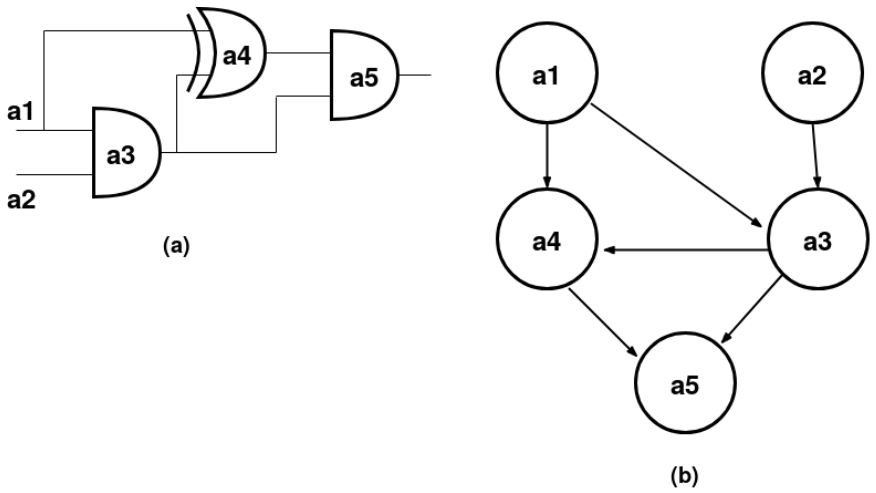


Figure 2.2: Fig. 2.2*a* shows an example Boolean circuit and Fig. 2.2*b* shows its corresponding DAG.

first to be traversed according to the graph's topological order, then node a_1 would not be revisited.

Consider the Boolean circuit shown in Figure 2.2a with two inputs, one output and three gates. Its corresponding DAG is shown in Figure 2.2*b*. The Boolean circuits we consider are acyclic, and therefore can be represented by a DAG. [RB12] proposes the use of cyclic topologies for Boolean circuits and shows that the complexity of implementing Boolean functions can be reduced with cyclic topologies than with acyclic topologies, therefore arguing that circuits can be designed with fewer gates if they contain cycles. However, in this thesis we assume acyclic topologies.

A DAG is defined by the relation $G = (V, E)$, with V being a set of nodes, $V = \{a_1, ..., a_n\}$ and E the set of edges $E \subseteq VxV$. The DAG shown in Figure 2.2b has five nodes and six edges. The direction of an arrow depicts a from-to relation in the directed graph. The number of incoming or outgoing edges of a node is known as its indegree or outdegree, respectively. For instance, from Figure 2.2b, node a_1 has indegree 0 and outdegree 2. The fanin or fanout of a graph is the maximum number of edges coming in or going out from its nodes respectively. We denote with $\Gamma_d(n)$ the set of all Directed Acyclic Graphs (DAGs) with n nodes and fanin and fanout d, where $n = u + v + k$. From Figure 2.2b, a_3 for example has fanin and fanout of 2.

For the purpose of this thesis, all Boolean circuits and their corresponding DAGs are assumed to have maximum fanin and fanout of 2. For Boolean circuits with gates with higher fanin, that is $d > 2$, the introduction of additional gates would reduce these gates to gates with fanin 2 (two inputs). This can be done based on Shannon's expansion theorem [Sha+49; Sch08]. For Boolean circuits with higher fanout, that is $d > 2$, copy gates are added in order to reduce the fanout of the circuit to 2. This is described in [Val76; KS16].

2.3 Secure Function Evaluation

Secure Function Evaluation (SFE) enables the computation of a globally known function which represents a Boolean circuit, $f(x_1, ..., x_n)$, to be computed based on the inputs of the communicating parties, $P_1, ..., P_n$. In the end, the parties compute $f(x_1, ..., x_n)$ without knowing any of the other parties' inputs. Fairplay [MNPS04] was the first implementation to achieve Secure Function Evaluation between two parties. Fairplay and FairplayMP [BNP08] were proposed for secure two-party and multi-party computations respectively.

Resembling C, Secure Function Definition Language (SFDL) is a language provided by the Fairplay framework to users in order to provide a high-level abstraction. Using this, users provide a high-level description of the function to be computed [BNP08]. The next step is to compile the given user specification (SFDL) into a low-level specification known as Secure Hardware Definition Language (SHDL). This specification is a Boolean circuit description as introduced in Section 2.1.2. The resulting SHDL file is then fed as input into two independent programs (in the case of two-party computation). Both parties then begin to communicate in order to achieve the final result using Yao's garbled circuit protocol [Yao86], which is a two-party computation protocol. For simplicity, in a two-party scenario where the globally known function is f, and the two parties both have x and y as inputs, then we have $f(x, y)$ as the result. The most prominent protocols used to achieve SFE are Yao's Garbled Circuit [Yao86] and the GMW protocol [GMW87], which we describe in detail in Section 2.3.2 and Section 2.3.3, respectively.

2.3.1 Oblivious Transfer

SFE is achieved by using special protocols which are coupled with Oblivious Transfer (OT) for sending and receiving messages obliviously. OT is a fundamental building block of many protocols that allows a sender to send two or more messages. The receiver is able to select one of the messages without the sender being able to learn which message was selected and without the receiver getting to know the content of the other messages. Assume Alice and Bob want to communicate with each other using OT. Alice has two inputs x_0 and x_1. Bob has an input bit b. Alice obliviously sends x_b to Bob using OT while Bob gets to know nothing about the other input x_{1-b}, neither does Alice get to know the input selected by Bob.

One of the most commonly used OTs is the above defined 1-out-of-2 OT where one of two sent inputs is selected by the receiver. This can be generalized to 1-out-of-n OT where one of the n sent inputs is selected by the receiver. OT is commonly used at the gate or input wire level for secure two-party and multi-party computations (which involves more than two communicating parties).

In a more practical scenario, the complexity of secure computations becomes high, hence more communication rounds are necessary. This comes at high costs of OTs thereby increasing the amount of time for computations. There exists so called OT Extensions which are used for

a_1	a_2	a_1 AND a_2
$k^0_{a_1}$	$k^0_{a_2}$	$E_{k^0_{a_1}}(E_{k^0_{a_2}}(k^0_{a_3}))$
$k^0_{a_1}$	$k^1_{a_2}$	$E_{k^0_{a_1}}(E_{k^1_{a_2}}(k^0_{a_3}))$
$k^1_{a_1}$	$k^0_{a_2}$	$E_{k^1_{a_1}}(E_{k^0_{a_2}}(k^0_{a_3}))$
$k^1_{a_1}$	$k^1_{a_2}$	$E_{k^1_{a_1}}(E_{k^1_{a_2}}(k^1_{a_3}))$

Table 2.1: Garbled table of an AND gate with input wires a_1 and a_2. k^i_w denotes the key of input wire w having bit i and $E_{k^i_w}$ denotes encryption using the key.

large-scale OT protocols that use a number of initial seed OTs (base OTs) [Bea96; ALSZ13; IKNP03]. The initial seed OTs are created using expensive public-key operations, and can be extended using cheap symmetric cryptographic operations to obtain many OTs. In [ALSZ15], the authors present an OT extension protocol for the setting of malicious adversaries that is more efficient and uses less communication than previous works.

2.3.2 Yao's Garbled Circuit

Yao's protocol is a two-party computation protocol which makes use of OT for exchanging messages between the computing parties [Yao86]. It is used for computing functionalities based on Boolean circuits. Yao's protocol has a constant number of rounds.

Let us assume Alice and Bob want to compute a function and that the functionality here is represented by a Boolean circuit. Alice assigns the garbled values which are random values to her corresponding wire inputs. The process is repeated for the other gates in the Boolean circuit. That is, k_0 for input bit 0 and k_1 for input bit 1. Alice is called the garbler. In addition, each gate is encrypted with the generated keys such that, for any given input wire x_n with its corresponding key k_n, the corresponding output wire o_n can be also generated. Table 2.1 shows the garbled table of a garbled AND gate. This garbled table resembles a Boolean truth table but instead of bits as inputs, we have a key associated with each input wire (for example $k^i_{a_1}$ for input wire a_1 with input bit i). In addition, the result column is replaced with the encryption of the output wire key using the two input keys, that is, $E_{k^i_{a_1}}(E_{k^j_{a_2}}(k_x))$ where k_x is the output wire key and i and j are input bits.

The garbled table has the structure of a logic gate's truth table and since the order can be guessed by an attacker, the garbled table is permuted to hide the order as well as to avoid any predictions on the part of the receiver, Bob. In the occurrence of encrypting an output gate type, the decryption of the output wire also has to be provided in the output translation table. However, Table 2.1 is not permuted as discussed for the sake of familiarity of order.

The whole encrypted circuit known as the garbled circuit is now sent to Bob as well as the keys generated for all of Alice's input wires. Bob is called the evaluator. In order to guarantee

the right garbled table entry is decrypted by Bob, Alice adds a bit for every input wire aside the generated key to inform which entry to decrypt. Depending on the number of Bob's inputs n, Alice and Bob run in parallel, n OTs for Bob to learn the remaining input keys. At this point, all keys have been obtained and Bob can retrieve the result.

Yao has a multiparty variant known as Beaver-Micali-Rogaway (BMR) which also has a constant number of rounds [BMR90]. [KS08b] introduces free XOR gates evaluation as optimization of Yao's Garbled Circuit protocol. That is, XOR gates are evaluated without the use of the associated garbled tables.

2.3.3 GMW Protocol

The protocol of Goldreich-Micali-Wigderson (GMW) was considered as less efficient compared to Yao's Garbled Circuit protocol [GMW87]. This is because Yao's Garbled Circuit is popular for its constant number of rounds. In [SZ13], the GMW protocol was optimized for two party computation by using means of load balancing and parallel processing of multiple gates.

GMW makes use of secret sharing of inputs. In the preprocessing or offline step, no inputs are needed yet. However, each computing party is assumed to know the number of AND gates in f. This determines the number of OT computations that have to be done during this preprocessing step. For each AND gate, a multiplication triple is first generated according to the relation $(a_1 \oplus a_2)(b_1 \oplus b_2) = (c_1 \oplus c_2)$ by both parties. This is done by means of one 1-out-of-4 OTs. [ALSZ13] shows how to generate these using two 1-out-of-2 OTs.

In the online phase, in a multiparty scenario with n computing parties, each party shares its input bits. With party A having input bit a_1, party A shares random bits $b_1, ..., b_{n-1}$ with all other $n-1$ parties (each party gets a different random bit) and sets its share as $a_1 \oplus \sum_{i=1}^{n-1} b_i$. This is such that XORing the individual shares generates the original input value a_1 of party A. That is, $b_1 \oplus b_2 ... \oplus b_n \oplus (a_1 \oplus \sum_{i=1}^{n-1} b_i) = a_1$ for n communicating parties. At the end, each party will have n shares, one from each other party including its own share.

For the sake of simplicity, henceforth, let us assume a two party computation scenario with Alice and Bob as the communicating parties. Now in the online phase, the next step is to evaluate the circuit. This is done in a gate by gate approach. Evaluating XOR gates is straightforward. This is carried out in a non-interactive manner since the computation can be done individually by each party. From sharing, Alice has received 2 inputs a_1 and b_1. Alice computes $c_1 = a_1 \oplus b_1$. Likewise, Bob computes $c_2 = a_2 \oplus b_2$. This is accurate because $a \oplus b = (a_1 \oplus a_2) \oplus (b_1 \oplus b_2) = (a_1 \oplus b_1) \oplus (a_2 \oplus b_2)$.

Evaluating AND gates involves one round of communication. To evaluate an AND gate, Alice would need in addition to the shares she possesses, a respective input share from Bob the value of which she does not know. The process is repeated this time by Bob to obtain the

Protocol	Yao's garbled circuit	GMW
Number of Parties	2-party and multi-party extension	Both 2-party and multi-party computations
Rounds	Constant number of rounds	offline: constant rounds. online: the number of rounds is linear in the depth of the circuit
Gates Evaluation	XOR and AND gates are evaluated differently with free XOR optimization. AND gates require offline and online symmetric key operations and offline communication.	XOR and AND gates evaluated differently. For AND gates: multiplication triples generated offline using OT. Requires 1 round of communication online. XOR gates are evaluated for free

Table 2.2: Comparison table of Yao's garbled circuit and GMW protocols.

missing shares needed for the gate evaluation. Now to the actual AND operation, both Alice and Bob compute $d_i = x_i \oplus a_i$ and $e_i = y_i \oplus b_i$ and send them to the other respective party. The multiplication triples a_i and b_i are used to mask the input shares x_i and y_i. They both then compute $d = d_1 \oplus d_2$ and $e = e_1 \oplus e_2$. Finally, Alice and Bob compute $db_1 \oplus ea_1 \oplus c_1 \oplus de$ and $db_2 \oplus ea_2 \oplus c_2$ respectively as the output shares of the AND gate [SZ13].

The number of rounds of the GMW protocol is $O(d)$ where d is the depth of the Boolean circuit. Table 2.2 enlists a few differences between Yao's garbled circuit protocol and the GMW protocol.

2.4 File Systems

In this section, we describe the file system. We begin by detailing general file system abstractions and types in Section 2.4.1. We then move on to describe file system tables in Section 2.4.2. Lastly we describe the file access service provided by the file system in Section 2.4.3.

2.4.1 File System Abstractions

Every computer, irrespective of the Operating System it runs or the purpose of the device, has a file system associated with it, so long as there is the need for storage of some sort for the functioning of the device. If this device has to store or record data as part of its functionality, then it most likely has a storage device embedded in it. The file system therefore acts as an abstraction on top of the storage device. Devices range from micro-controllers, smart phones, personal computers to super computers and many others. Taking a personal computer as

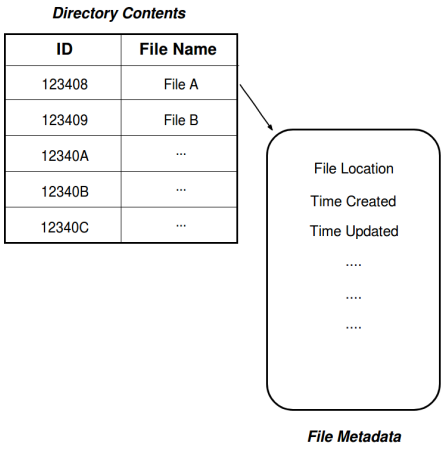


Figure 2.3: Figure of sample File System Table and File Metadata

an example, the view of the Operating System's file system and directories presented to users and applications is just but an abstraction. In computers, the hard drive is a primary example.

Disk partitions divide a hard drive into two or more regions or partitions. A partition is a region with content on a hard drive that is managed by the Operating System. Each partition is represented visually by the Operating System as one or more logical drives which occupies part of the whole hard drive or disk. It is on these logical drives that the file system and files are created. The logical drive also contains information about the file system, such as the file system type and also the boot record.

2.4.2 File System Tables

Files and directories on a hard drive make up the file system objects. The file system stores data and metadata for files and directories. The file system table stores the attributes and location(s) of the data of the files and directories. Metadata includes location on disk, last updated time, time created, user id, number of links and file mode as shown in Figure 2.3. There are two main types of metadata, in particular block allocation bitmaps and data structures, that keep track of files and the disk blocks to which they belong [Reu06].

In block allocation, available disk space is divided into groups of fixed block sizes. The allocation of block size to each fixed group is done in a dynamic fashion.

The data structures that keep track of files and the disk blocks to which they belong are stored in the file system table. A list of blocks that belongs to each file is maintained in the table.

2.4.3 File Access

Files can be opened for either sequential or direct access.

In sequential access, the current file pointer position moves in a linear manner. That is, the file is accessed from the beginning to the end. This is done in a byte sequence order. However, it is possible to reposition the current file position pointer by means of a system call, for instance a system call to reopen the same file or a system call to write data to the same file. Sequential access works best when our aim is to process files consisting only of text, such as the files created with a typical text editor. That is, files in which data is not divided into a series of records. One major problem with sequential access is preprocessing or going through all previous lines (in the case of a text file) prior to the point of interest. In this work, we open files for sequential access.

In direct access, the current file pointer position can be repositioned to any point in the file. This is random as opposed to the linear byte sequence approach used in sequential access. File systems that are optimized for use with databases are accessed in terms of records rather than a byte sequence (that is direct, based on an index for instance, rather than sequentially).

3 Existing Universal Circuit Constructions

3.1 Private Function Evaluation

Private Function Evaluation involves two or more parties who would like to compute a function where one party P_1 has the function $f(x_1, ..., x_n)$, to be computed represented as a circuit C_f, and the other party(ies) $P_2, ..., P_n$ provide the input(s) to the function. For simplicity, let us assume there are only two parties involved in this computation. The goal is to enable in the end, both parties to compute the function $f(x)$ without knowing the input of the other respective party. In other words, P_1 provides f and P_2 provides x and together they compute $f(x)$, without P_1 learning anything about the input x and P_2 learning anything about the function f. [MS13] presents a general framework for Private Function Evaluation for both Boolean and arithmetic circuits. They further improve this result to achieve security against malicious adversaries in [MSS14]. These protocols have better computation complexities than the method studied in this work, but they have worse communication complexities, as shown in [GKS17].

3.2 Universal Circuits for Representing Functions as Data

Secure Function Evaluation (SFE) allows two parties to securely evaluate a function on their respective private inputs, without revealing anything other than the result to the other party (cf. Section 2.3). Being able to keep the respective inputs of both parties x and y private with SFE, the problem of evaluating function f privately is transformed into the problem of finding a globally known structure that can simulate functions based on some data that is provided by one party, and the input that is provided by the other party, in other words representing the function to be evaluated as input data. The idea here is to achieve Private Function Evaluation by using Secure Function Evaluation. In this sense, the need to make the function f private introduces the necessity to maintain a publicly known entity. In this publicly known structure, the private function f is provided by one party and the private input x provided by the other party in order for both parties to compute $f(x)$. The globally known structure is a Universal Circuit (UC).

Valiant proposed a method of constructing Universal Circuits based on the use of so-called Edge-Universal Graphs (EUGs) [Val76]. He lays down the underlying foundations for the construction of Universal Circuits: representing circuits as DAGs, finding an Edge-Universal

Graph (EUG) and the method with which it can represent this graph and transforming it into a UC. He provided two theoretical ways of Universal Circuit construction based on two recursive methods: 2-way split and 4-way split, where the recursive structure consists of either 2 or 4 sub-structures respectively. Valiant's UC construction has asymptotically optimal size $O(n \log n)$ [Val76].

Another independent UC implementation was proposed by Kolesnikov and Schneider with size $0.75k \log_2^2 k + 2.25k \log_2 k + k \log_2 u + (0.5k + 0.5v) \log_2 v$ [Sch08; KS08a]. Actual implementations of Valiant's UC construction were realized in two independent works [KS16; LMS16] using the 2-way construction. For the sake of more efficient and optimized Universal Circuit constructions, Günther, Kiss and Schneider realized an implementation of the 4-way split Universal Circuit construction with size of $4.75n \log_2 n$ in [GKS17]. In that same work, they also discuss the possibility for a more efficient Universal Circuit based on a novel 3-way split, which is then proven to be less efficient than the 2- or 4-way split methods. We detail these methods in the next section.

3.3 Valiant's Universal Circuit Construction

In this section, we begin with some background of Valiant's UC construction that is specific for Edge-Universal Graphs in Section 3.3.1. This is followed by the construction methods including the 2- and 4-way split methods in Section 3.3.2. The latter is the construction on which our implementation of improving the scalability of UC for large-scale Private Function Evaluation is based. Finally in Section 3.3.3 we discuss other UC constructions.

3.3.1 Edge-Universal Graphs

We consider a DAG with a set of nodes $V = \{a_1, ..., a_n\}$ and a set of edges $E \subseteq VxV$. The DAG has size n with fanin and fanout d. This means that the DAG is in $\Gamma_d(n)$ (cf. Section 2.2). Having a set of DAGs $D = \{d_1, ..., d_k\}$ each with an arbitrary size n, one can construct a generic graph or structure so that we can map each and every graph contained in this set D to this generic graph. In other words, all the graphs can be obtained from this generic graph by a defined mapping. An EUG is such a universal representation of all directed acyclic graphs of a given arbitrary size n. This mapping defines a unique path on every DAG in D with size n to its corresponding EUG $U_n(\Gamma_2)$. This mapping is such that, for every edge $e \subseteq E$ in the DAG, there exists a unique path on the EUG.

The Structure of an EUG

Figure 3.1 shows two graphs that are able to represent DAGs of size 5 and fanin and fanout of one and two in Figures 3.1a and 3.1b, respectively. These graphs are edge-universal graphs denoted by $U_5(\Gamma_1)$ and $U_5(\Gamma_2)$. Both graphs contain poles $P = \{p_1, ..., p_5\}$ represented as

squares and special nodes (dark circles). In Figure 3.1a, the EUG, denoted as $U_5(\Gamma_1)$ is of size 12, that is five poles plus seven special nodes. The poles are the original nodes of the DAG which can be derived from this EUG. The size is greater than its corresponding DAG because of the special nodes that are added to enable multiple paths so that all possible combinations of paths are covered. Poles map to the actual nodes of a DAG, whereas the special nodes of the EUG help with multiple routing. There also exists a difference between fan-in/out of the EUG and the set of DAGs it represents. As depicted in Figure 3.1a, the special node marked with the letter *A* shown in the picture contains two incoming and outgoing edges, that is fanin and fanout of two. This implies that, for a DAG of a given size with fanin and fanout one, its corresponding EUG may have nodes with two incoming and/or outgoing edges, therefore, the special nodes may have fan-in/out of two. However, poles have fanin and fanout of one in the case of Figure 3.1a.

Ideally, we require $U_n(\Gamma_2)$ in order to represent DAGs with fanin and fanout of 2. Hence, all nodes in the graph $U_n(\Gamma_2)$ will have fanin and fanout of 2 irrespective if the node is a pole or a special node. For a given EUG $U_n(\Gamma_1)$, we obtain $U_n(\Gamma_2)$ by combining two $U_n(\Gamma_1)$s. This graph is denoted by $U_5(\Gamma_2)$ and is obtained by merging two $U_5(\Gamma_1)s$ as shown in Figure 3.1b. Specifically in our case, the graph has fanin and fanout of 2 and 14 special nodes. In the EUG $U_5(\Gamma_2)$ formation procedure, poles of the two $U_n(\Gamma_1)$ graphs are merged and therefore the number of poles remains unchanged, only their fanin and fanout is increased from one to two. In addition, new special nodes are introduced into the structure. They help each pole in achieving fanin and fanout of 2 and they also help with multiple routing of edges $E \subset VxV$ in the DAG. Hence, an EUG for $\Gamma_2(n)$ has double the number of special nodes compared to an EUG for $\Gamma_1(n)$, since it is merged from two EUGs for Γ_1. We look into the concrete constructions proposed by Valiant in [Val76] and improved in [KS16; LMS16; GKS17].

3.3.2 Valiant's Universal Circuit

In this section, we begin by discussing the stages of obtaining $U_n(\Gamma_2)$ for all $\Gamma_2(n)$. We move on to discuss Valiant's recursive universal graph constructions proposed in [Val76].

The aim is to construct a UC from which circuits of a specific size can be derived. Having an EUG, we can derive DAGs of a certain size n. By mapping a circuit that has u inputs, v outputs and k gates to a DAG of size $n = u + v + k$, we can make use of $U_n(\Gamma_2)$ (built out of two $U_n(\Gamma_2)s$) to construct a UC that can simulate circuits of that given input, output and gate size.

For a given graph $G = (V, E)$ representing a Boolean circuit $C^k_{u,v}$ with size $n = u + v + k$, there are three stages involved in the construction of its UC: firstly a derivation of $U_n(\Gamma_1)$; followed by a derivation of $U_n(\Gamma_2)$ from merging two $U_n(\Gamma_1)$s; then finally constructing the actual UC.

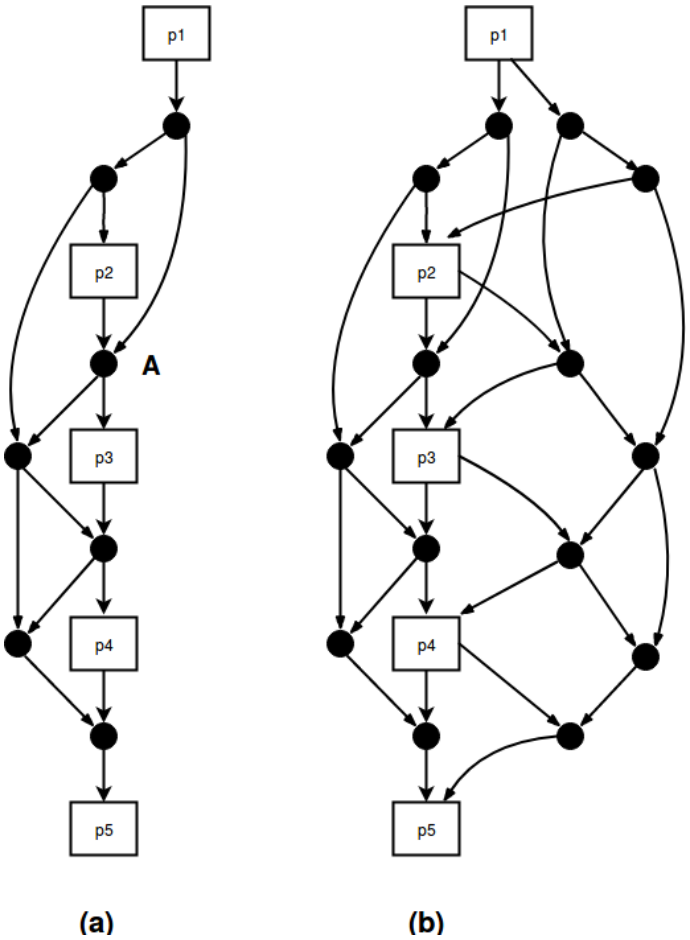


Figure 3.1: **(a)** Universal graph $U_5(\Gamma_1)$ that can represent DAGs of size 5 with fanin and fanout 1 ($\Gamma_1(5)$) and **(b)** Universal graph $U_5(\Gamma_2)$ that can represent DAGs of size 5 with fanin and fanout 2 ($\Gamma_2(5)$).

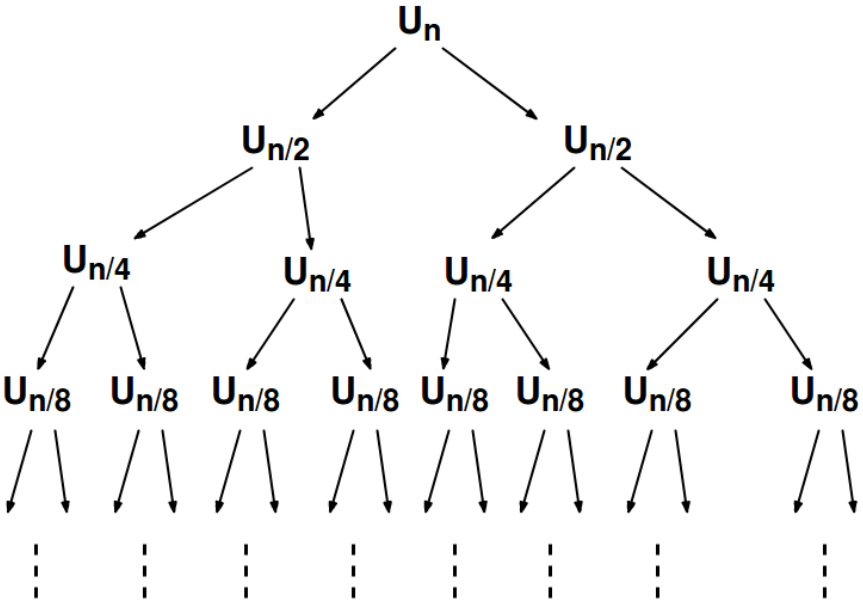


Figure 3.2: An overview of the recursive 2-way split EUG construction for $\Gamma_1(n)$.

Constructing $U_n(\Gamma_1)$

Valiant proposed EUG constructions which serve as the recursion bases for Γ_1 graphs in [Val76]. For the sake of simplicity, we denote by U_n the $U_n(\Gamma_1)$ graphs. From these recursion bases, all other EUGs U_n of a given size n can be constructed. These recursion bases are U_1, U_2, U_3 and U_4. U_1 is a graph with a single pole. U_2 and U_3 are graphs with two and three connected poles respectively. U_4 has alternating poles and nodes from top to bottom, hence 3 additional nodes. Valiant also proposed hand-optimized structures for U_5 and U_6, which is shown in [Val76; KS16]. To derive U_n greater than the aforementioned base cases, a recursive construction algorithm is utilized. The construction algorithm is recursive such that it halts when any of the base cases is reached. Valiant proposed two methods of EUG construction: the 2-way and 4-way split methods. Figure 3.2 shows the 2-way split construction from a high-level perspective. From this figure, the construction graph is structured like a binary tree where the parent or root node is the EUG of size n to be constructed, that is, U_n. The last leaf node is one of the base case EUG graphs.

It is evident at each graph level that each EUG consists of two new EUGs so called subgraphs. Each subgraph U_n contains its own set of poles $V = \{p_1, ..., p_n\}$. Let the first two subgraphs $U_{n/2}$[1] be represented by $U^1_{n/2}$ and $U^2_{n/2}$ respectively, where $U^1_{n/2} = \{q_1, ..., q_{[\frac{n-2}{2}]}\}$ and $U^2_{n/2} = \{r_1, ..., r_{[\frac{n-2}{2}]}\}$. $U^1_{n/2}$ and $U^2_{n/2}$ are the poles of the next recursion step, which will in turn be used to create the next subgraphs $U^{11}_{n/4}$ and $U^{12}_{n/4}$ (for $U^1_{n/2}$) and $U^{21}_{n/4}$ and $U^{22}_{n/4}$ (for $U^2_{n/2}$) in the following recursion step. This process is repeated until the number of poles is same as

[1] We denote $U_{(n-2)/2}$ with $U_{n/2}$ for the sake of simplicity of notation.

either of the recursion bases. The 2-way split has since its proposal by Valiant, been brought to practice by [KS16] and [LMS16].

Valiant's 4-way split method is implemented in [GKS17]. Analogously to the 2-way split construction method, there are 4 subgraphs at each EUG-split level in the 4-way split. Let the first four subgraphs be represented by $U^1_{n/4}$, $U^2_{n/4}$, $U^3_{n/4}$ and $U^4_{n/4}$ where $U^1_{n/4} = \{q_1, ..., q_{[\frac{n-4}{4}]}\}$, $U^2_{n/4} = \{r_1, ..., r_{[\frac{n-4}{4}]}\}$, $U^3_{n/4} = \{s_1, ..., s_{[\frac{n-4}{4}]}\}$ and $U^4_{n/4} = \{t_1, ..., t_{[\frac{n-4}{4}]}\}$. Four subgraphs are created at each subgraph level per EUG until a base case is reached. This is equivalent to taking every second layer from the construction depicted in Figure 3.2. In the 4-way split construction method, there are different types of underlying construction blocks used so called the head, body and tail blocks (we give further details in Chapter 4). Unlike in the 2-way split where there are 1 or 2 alternating poles and special nodes between subgraphs in the main skeleton (that is, after every second alternating pole and special node, the graph splits into two new subgraphs), the 4-way split could have 1, 2, 3 or 4 poles per block depending on the size and different number of special nodes depending on the position of the block in the construction.

Having an EUG with number of poles n, we should be able to map every DAG of size n in $\Gamma_1(n)$ into the given EUG. This mapping is called edge-embedding. For a given DAG $G = (V, E)$ with nodes $V = \{a_1, ..., a_n\}$ and edges $E \subset VxV$, all nodes are mapped one-to-one to the poles of the EUG $P = \{p_1, ..., p_n\}$. In terms of edges, every edge (i, j) in the DAG is mapped to a path that connects the poles p_i and p_j. The mapping should be such that there exists edge-disjoint paths between poles i and j as well as subsequent poles. In other words, every path between two poles p_i and p_j mapped into the EUG should be unique and an edge $E \subseteq VxV$ is used only for one path.

Constructing $U_n(\Gamma_2)$

As pointed out earlier, the next stage of the Universal Circuit construction is to create a structure from which all DAGs of fanin and fanout 2 and size n ($U_n(\Gamma_2)$) can be derived. From Figure 2.2, our aim would be to obtain $U_5(\Gamma_2)$ (from Figure 3.1b) to represent this DAG.

We derive $U_n(\Gamma_2)$ by merging two $U_n(\Gamma_1)$s (described earlier in this section) by their poles $P = \{p_1, ..., p_n\}$. As mentioned in Section 3.3.1, the number of poles in the new structure remains unchanged and the special nodes introduced help the poles in achieving fanin and fanout of 2. All poles and nodes therefore have fanin and fanout of 2 after merging, which enables the embedding of twice as many edges to paths in the EUG than before.

To define the unique paths of edges of the DAG unto the universal graph, a method so called coloring is employed. This method is based on König-Hall's theorem [LP09]. A given DAG denoted as $G = (V, E)$ belonging to $\Gamma_2(n)$ can be divided into two graphs with fanin and fanout of 1 $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ with same n number of nodes [Val76]. Hence, the two set of edges $E_1, E_2 \subseteq VxV$ can be each edge-embedded into two $U_n(\Gamma_1)$ (which

together build up $U_n(\Gamma_2)$). A topological ordering of the nodes of the derived $U_n(\Gamma_2)$ has to be ensured. We edge-embed the Γ_2 graph $G = (V, E)$ into the derived $U_n(\Gamma_2)$ EUG, such that every edge $e \in E \subseteq VxV$ is mapped to a unique path. We discuss the edge-embedding method in Chapter 4.

Constructing UC

To actually construct the UC, we need to program the $U_n(\Gamma_2)$ EUG according to the edge-embedding of the DAG. For the given DAG with size $n = u + v + k$, k gates are programmed as universal gates and all other nodes are programmed either as so-called X or Y switches. We discuss this in detail below.

Valiant's UC construction has asymptotically optimal size $O(n \log n)$ [Val76]. Concretely, the 2-way split UC construction has complexity $\sim 5n \log n$, while the complexity of the 4-way split construction is $\sim 4.75n \log n$ [GKS17].

In this section, we begin by introducing so-called modular block types of the EUG as detailed in [GKS17].

For a given Boolean circuit $C^k_{u,v}$ to be edge-embedded, k gate nodes are programmed as universal gates and the remaining nodes of the universal circuit are programmed as either X or Y switches.

Valiant's UC construction is applied to acyclic circuits with logic gates that realise one of the Boolean functions $\{0,1\}^2 \rightarrow \{0,1\}$ [Val76]. Valiant introduced universal gates which accept two inputs x_0, x_1 and $c = \{c_0, c_1, c_2, c_3\}$ which are the input wires and control bits respectively [Val76]. With four control bits, every universal gate $U(x_0, x_1; c_0, c_1, c_2, c_3)$ accepts two inputs and has 16 possible functionalities it can represent for every fixed assignment of control bits $c = \{c_0, c_1, c_2, c_3\}$. This is shown in Equation (3.1).

$$y = c_0 x_0 x_1 + c_1 \overline{x_0} x_1 + c_2 x_0 \overline{x_1} + c_3 \overline{x_0 x_1} \tag{3.1}$$

As mentioned before, the remaining nodes are programmed as either X or Y switches. X switches have two inputs x_0, x_1 and have one control bit c to define their behavior. X switches denoted by $(y_0, y_1) = X(x_0, x_1; c)$ return as outputs, the two inputs either in the same order or reversed order. Equation (3.2) shows its functionality.

$$y_0 = \overline{c} x_0 \oplus c x_1 \qquad\qquad y_1 = c x_0 \oplus \overline{c} x_1 \tag{3.2}$$

Y switches are also defined by two inputs and one control bit. Y switches denoted by $y = Y(x_0, x_1; c)$ return as output either one of their two inputs as shown in Equation (3.3).

$$y = (x_0 \oplus x_1)c \oplus x_0 \tag{3.3}$$

From the above equations, it is observed in [KS16] that universal gates can be implemented using six XOR gates and three AND gates. X and Y switches or gates are implemented using one AND gate and three XOR and two XOR gates, respectively.

3.3.3 Other Universal Circuit Constructions

In this section, we discuss other UC constructions of which some are based on Valiant's UC construction. We begin with UC construction for smaller circuits proposed in [KS08a]. This is followed by the generic k-way split and the hybrid UC constructions, that are extensions of Valiant's UC construction. Finally, we discuss Universal Arithmetic Circuits (UAC) as proposed in [LMS16].

UC Construction for Smaller Circuits

[KS08a] introduced a universal block U_k with inputs and outputs for every logic gate G_i in a Boolean circuit with size k. For each gate G_i, there exists a simulation gate G_i^{Sim}. Now to construct the UC, an input selection block $S^u_{2k\geq u}$ which permutes and selects maximum $2k$ inputs from the actual circuit to U_k is used. This also hides the input wiring by assigning duplicates to the input selection block. Additionally, an output selection block $S_v^{k\geq v}$ maps k outputs from U_k to the actual outputs of the UC. This combined entity yet to be programmed is known as a programmable block and is denoted as B^u_v, where v is the number of outputs from the output selection block which is essentially k. The topological order of the gates is then ensured such that for $(G_i, G_j) \in C^k_{u,v} \Rightarrow \eta(G_i) < \eta(G_j)$. In other words, if G_i comes before G_j, then G_i should have no inputs that are outputs of G_j. As in the 2-way split UC construction, two $U_{k/2}$ blocks are then combined to form U_k.

Kolesnikov and Schneider propose UC construction which resulted in up to 50% reduction in size as compared to Valiant's UC construction [KS08a]. This reduction is due to lower constant factors for small circuits with particular interests for Private Function Evaluation (PFE) applications of sizes up to 1000. However, Valiant's UC is asymptotically better. For a circuit $C^k_{u,v}$ with u inputs, v outputs and k gates, Valiant's 2-way UC has size $(5k\log k + 9.5u + 9.5v)\log k + O(k)$ compared to $1.5k\log^2 k + 2.5k\log k + (u+2k)\log u + (k+3v)\log v + O(k)$ [KS08a].

k-Way Split UC Construction

Lipmaa et al. proposed a generalization of Valiant's construction known as k-way split for UC construction in [LMS16]. A structure known as a supernode is constructed from two permutation blocks (networks) and an inner block between the permutation networks. The size of the permutation and inner blocks varies with k, where k is the number of sub-structures in the recursion to be used in the UC construction. The first permutation network helps to map any permutation of inputs to nodes and likewise, the latter permutation network maps any permutation of nodes to output. Both permutation networks have k inputs and outputs. Lipmaa et al.'s proposed k-way UC construction for $k \in \{2, 4\}$ also has decreased the total size compared to Valiant's UC construction.

Hybrid UC Constructions

Kiss and Schneider [KS16] propose a hybrid UC construction which combines techniques from constructions from both Valiant and Kolesnikov and Schneider [Val76; KS08a]. The hybrid approach is more efficient for circuits with many inputs and outputs, where the number of inputs and/or outputs is linear in the number of gates [KS16].

For a UC with u inputs, v outputs and k gates, three scenarios were considered: a constant I/O case where u and v are both constants; a case of many inputs where $u \sim k$; and a maximal I/O case where $u \sim 2k$ and $v \sim k$. As it turns out, it is advisable to use the hybrid UC construction in cases of many inputs and maximal I/O as this yields smaller UC sizes compared to Valiant's UC construction with the same scenarios [KS16].

Günther, Kiss and Schneider propose a generic hybrid UC construction that combines multiple k-way UC constructions [GKS17]. For instance for values $k = \{2, 4\}$, their hybrid UC construction combines 2-way and 4-way split methods. According to their evaluation, the resulting hybrid UC has the smallest UC size compared to the 2-way and 4-way UC constructions for any given input circuit [GKS17].

Universal Arithmetic Circuits

Lipmaa, Mohassel and Sadeghian [LMS16] extended Valiant's UC construction to construct Universal Arithmetic Circuits (UAC). For a given DAG with fan-in/out of 2, its $U_n(\Gamma_2)$ is implemented using arithmetic operations instead of using binary gates. The UAC is constructed with size of at most $O(n) + 5L$ where L is the number of nodes in the constructed UAC (with the exception of poles) and n is the number of nodes of the Boolean circuit [LMS16]. They claim that their UAC construction has the same asymptotic complexity as Valiant's UC construction for Boolean circuits.

In [LMS16]'s Universal Arithmetic Circuit (UAC) construction, the arithmetic universal gates are programmed differently from Valiant's Boolean UC construction. Seven different functions are required in addition to three control bits $c = \{c_0, c_1, c_2\}$. The remaining nodes are programmed as X or Y switches. For instance, X switches are programmed as: $(c, x_0, x_1) \rightarrow ((1-c)x_0 + cx_1,\ cx_0 + (1-c)x_1)$, and Y switches are programmed analogously. More details on the functions and their implementation can be found in [LMS16].

3.4 Limitations of Existing Universal Circuit Implementations

Though Valiant's Universal Circuits have efficient implementations, all these implementations have memory consumption as bottleneck since the whole Universal Circuit is stored and programmed in memory. In other words, the largest circuit that can be computed is directly influenced by the available memory. Due to this limitation, the need arises to address

this issue. The method could be slower, since in most applications, the UC generation happens in a precomputation step, but should not store more than $O(n)$ information at any moment in time in memory (as opposed to the current $O(n \log n)$ information, for example in Günther et al.'s implementation), in order to allow for the computation of very large circuits.

Methods and mechanisms for proactive memory management, such as caching, fail at this case because, first of all, computations involved in the Universal Circuit construction are linear and therefore cache hits may be very low. Moreover, for the construction involving very large circuits, a large amount of addressable memory is needed for successful computation. Virtual memory automatically swaps some data from memory to disk for later use, but this occurs when the allocated memory of a program is almost used up. It is only a matter of time before a bad_alloc error is thrown if the program continues using up memory. We discuss this further in our evaluation in Chapter 6. Of course, the above indicated flaw is from a memory-only method's perspective. In relation to Private Function Evaluation, there have not been any implementations with regards to solving the boundary presented by the use of memory.

When it comes to scaling Universal Circuits, not much has been achieved because of the fact that there has been only a few implementations realized, as discussed above. It is therefore the purpose of this thesis to explore methods for scaling Universal Circuits for Private Function Evaluation. The method discussed in this thesis makes use of both memory and the file system to achieve scalability of UC construction for very large Boolean circuits.

4 Design

In this chapter, we detail the design of our improvements to the 4-way split UC construction implemented in [GKS17] for large-scale private function evaluation. We introduce further details about the 4-way split UC construction including its block types and a few algorithms in Section 4.1. This leads us to the next section on refining the blocks covered in Section 4.1 to improve on the UC construction in Section 4.2. Finally in Section 4.3, we discuss the design of the scalable programming of the UC.

4.1 Valiant's 4-way Universal Circuit Construction

In this section, we begin by introducing so-called modular block types of the EUG as detailed in [GKS17].

Valiant's 2-way split UC construction was implemented in [KS16; LMS16]. Valiant's 4-way UC construction was made modular in the implementation in [GKS17].

With motivation from [KS08a; KS16], Günther, Kiss and Schneider define three types of programmable blocks for UC construction in [GKS17]. The general body block is denoted as $B_{k,k}^{P(l)}$ where l is the number of additional hidden inputs (the program bits) in the programmable block and k is the input or output permutation block. For a DAG representing a circuit $C_{u,v}^{k}$, we define a pole as either of u, v or k nodes and is represented as big circles as shown in Figure 4.1. X and Y switches are represented as smaller circles also known as special nodes. The body block has four poles and fifteen special nodes. The body block has eight so-called recursion points in total which are emphasized as squares. Recursion points are the poles of the subgraphs in the next recursion step. Figure 4.1a shows the body block.

[GKS17] refines the general body block to design Head and Tail blocks. These are special programmable blocks that only occur at the top and bottom of a chain of blocks, respectively. The Head block has no upper recursion points and is denoted as $B_{0,k}^{P(l)}$. The lack of upper recursion points is due to the fact that it is the beginning of a chain of blocks and therefore has no inputs being passed to the block. Figure 4.1f shows the head block. The Tail block has no lower recursion points and is denoted as $B_{k,0}^{P(l)}$. Analogously to the head block, the tail block has no lower recursion points because it ends the blocks' chain and therefore maps to the actual outputs of the circuit. The tail block can have either one, two, three or four poles according to ($n \mod 4$) where $n = u + v + k$. When $n \mod 4 = 0$, the number of poles in

the tail block is 4. Figures 4.1b, 4.1c, 4.1d and 4.1e show tail blocks with one, two, three and four poles respectively.

In addition to the poles in each block, there are hidden inputs, the number of which is denoted as l. These hidden inputs enable multiple routing during edge-embedding of the UC. The value of l depends on the type of block, that is, for body blocks it is $l = 15 + 4 \times n_{ug}$, for head blocks $l = 7 + 4 \times n_{ug}$, where n_{ug} is the number of universal gates in the body block.

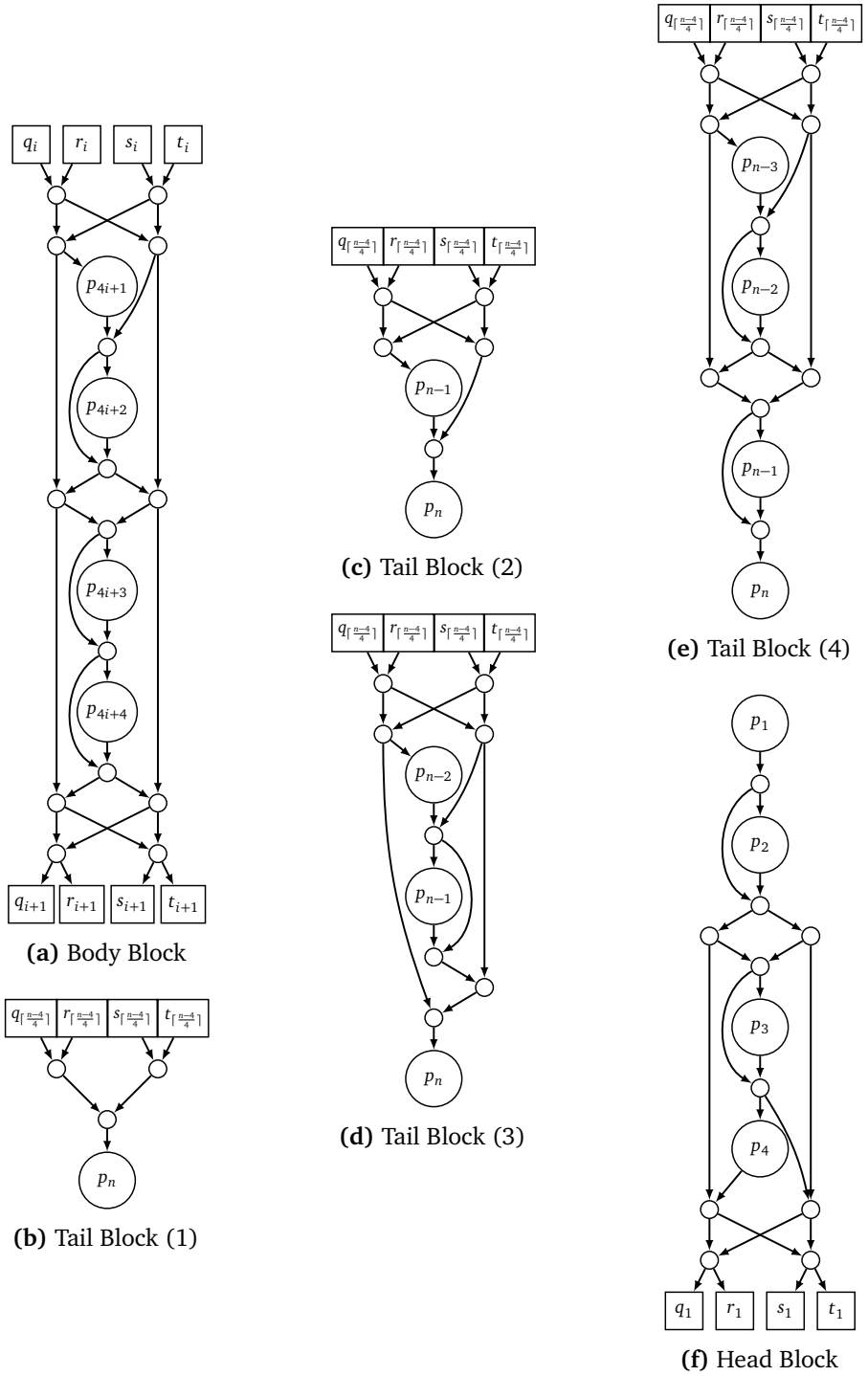
Actually, the head block has 10 nodes (apart from poles) as we can observe in Figure 4.1f. However, three of these nodes are special nodes identical to reversed Y switches with one input and two outputs. For a given input to a reverse Y switch, the outputs can be duplicated (inputs) and therefore we do not need to present it as an additional gate in the circuit level, we can use two wires instead. Hence, we require only 7 hidden inputs in the number of hidden inputs for the head block. Depending on the number of poles in the Tail block, l for the tail block is calculated as $l = n_{nwp} + 4 \times n_{ug}$, where n_{nwp} is the number of nodes apart from poles.

So far we have not described the recursion base block. The recursion base block, which is constructed in a so-called recursion step subgraph contains the recursion base cases with four or less nodes. Figure 4.2 shows a recursion base with four poles (recursion points) and three nodes in the subgraph. As discussed earlier in Section 3.3, the recursion points are the poles in the next recursion step subgraph. Given a DAG with $n = u + v + k = 20$, we can construct $U_n(\Gamma_1)$ with five blocks: one head, one tail and three body blocks. It becomes evident that there are four recursion point sets, each with 4 poles, that is, $\{q_1, ..., q_4\}$, $\{r_1, ..., r_4\}$, $\{s_1, ..., s_4\}$ and $\{t_1, ..., t_4\}$. We can therefore construct four recursion base graphs with four poles and three nodes in the subgraph.

For a given EUG $U_n(\Gamma_1)$, the UC construction includes at least a head and a tail block. For $n \geq 9$, there exists one or more body blocks in the UC construction. As discussed earlier in Section 3.3.2, two $U_n(\Gamma_1)$ graphs are combined to form $U_n(\Gamma_2)$. This chain of blocks from the head block to the tail block represents $U_n(\Gamma_1)$ and hence, two chains of blocks are combined to fully construct the UC.

4.2 Scalable Universal Circuit Generation

In this section, we detail our per-block approach for UC generation in Section 4.2.1. Thereafter, we discuss our per-block topological ordering in Section 4.2.2. For the subgraphs in the UC, we discuss a recursive method of generating subgraph nodes in Section 4.2.3.



(a) Body Block

(b) Tail Block (1)

(c) Tail Block (2)

(d) Tail Block (3)

(e) Tail Block (4)

(f) Head Block

Figure 4.1: Figure from [GKS17]. (a) shows Valiant's 4-way split EUG construction [Val76]. (b)-(e) show tail block constructions for different number of poles (denoted in brackets). (f) shows head block construction.

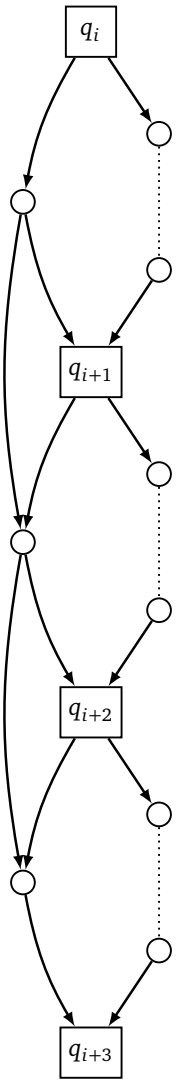


Figure 4.2: shows the recursion base of recursion point set $Q = \{q_i, q_{i+1}, q_{i+2}, q_{i+3}\}$ with 4 recursion points (poles of the subgraph) and 3 subgraph nodes.

4.2.1 Per-Block 4-way Generation

In the implementation presented in [GKS17], the Universal Circuit structure is held in memory, which poses an obvious bound on the maximum UC size. The maximum UC that can be constructed is limited to the available memory capacity on the computing machine or cluster. In [GKS17] an evaluation was run with an upper bound of $n = 2 \cdot 10^5$ nodes. Our aim is to overcome this boundary by using the file system for the UC construction.

Our Construction

To avoid the limitations imposed by memory-dependent implementations of Valiant's UC construction, we introduce a per-block UC construction approach. The underlying idea behind this approach is to generate a block one at a time starting from the head block until the tail block of the chain of blocks, never keeping more than one block and its corresponding subgraph nodes in memory. In addition, the required recursion step subgraph nodes per-block are constructed according to their rightful positions in the actual chain of blocks. After the requisite nodes (including recursion step subgraph nodes) have been created, they are then topologically ordered and the necessary nodes' information are output to a file, after which the created nodes are destroyed.

As discussed in Section 3.3.2, $U_n(\Gamma_2)$ is formed from two instances of $U_n(\Gamma_2)$. From here on, we refer to these as the left and right subgraphs of the DAG. From each subgraph, there are smaller subgraphs known as recursion point subgraphs (or recursion step graphs) that help during edge-embedding as we will recapitulate in Section 4.3. From Figure 3.2, we see that these recursion point subgraphs for a 2-way split UC are the set of EUGs $\{U_{n/2}, U_{n/4}, U_{n/8}, ...\}$ until the recursion base is reached. For a 4-way split recursive UC construction, the recursion point subgraphs are the set of EUGs $\{U_{n/4}, U_{n/16}, ...\}$. Henceforth in this chapter, we term U_n as the outer skeleton (also known as the recursion step graph) and $\{U_{n/4}, U_{n/16}, ...\}$ as subgraphs (also known as recursion step subgraphs).

The subgraphs emerge from recursion points (starting from the head block's lower recursion points) and end at the last set of recursion points at the end of the chain of blocks (upper recursion points of tail block). The last recursion step subgraph nodes at each body block of the outer skeleton have outgoing wires directed to the lower recursion points of the outer skeleton's body block as well as the next node(s) of the subgraph. Head and tail blocks only have lower and upper recursion points respectively. The head and tail blocks therefore mark the beginning and end of subgraphs respectively. In other words, the first subgraph nodes are created after the head block (from its lower recursion points) and the last subgraph nodes end before the tail block (before its upper recursion points). Therefore the structures we introduce only apply to body blocks of the outer skeleton (Figure 4.3).

For a given $U_n(\Gamma_1)$, there are $\lceil \frac{n-4}{4} \rceil - 1$ body blocks. For a given body block $B_{4,4}^{P(l)}$, we introduce two sets of subgraph nodes $S = \{s_q, s_r, s_s, s_t\}$ and $T = \{t_q, t_r, t_s, t_t\}$ where S is the set of last subgraph nodes (from the last constructed body block) for each of recursion points

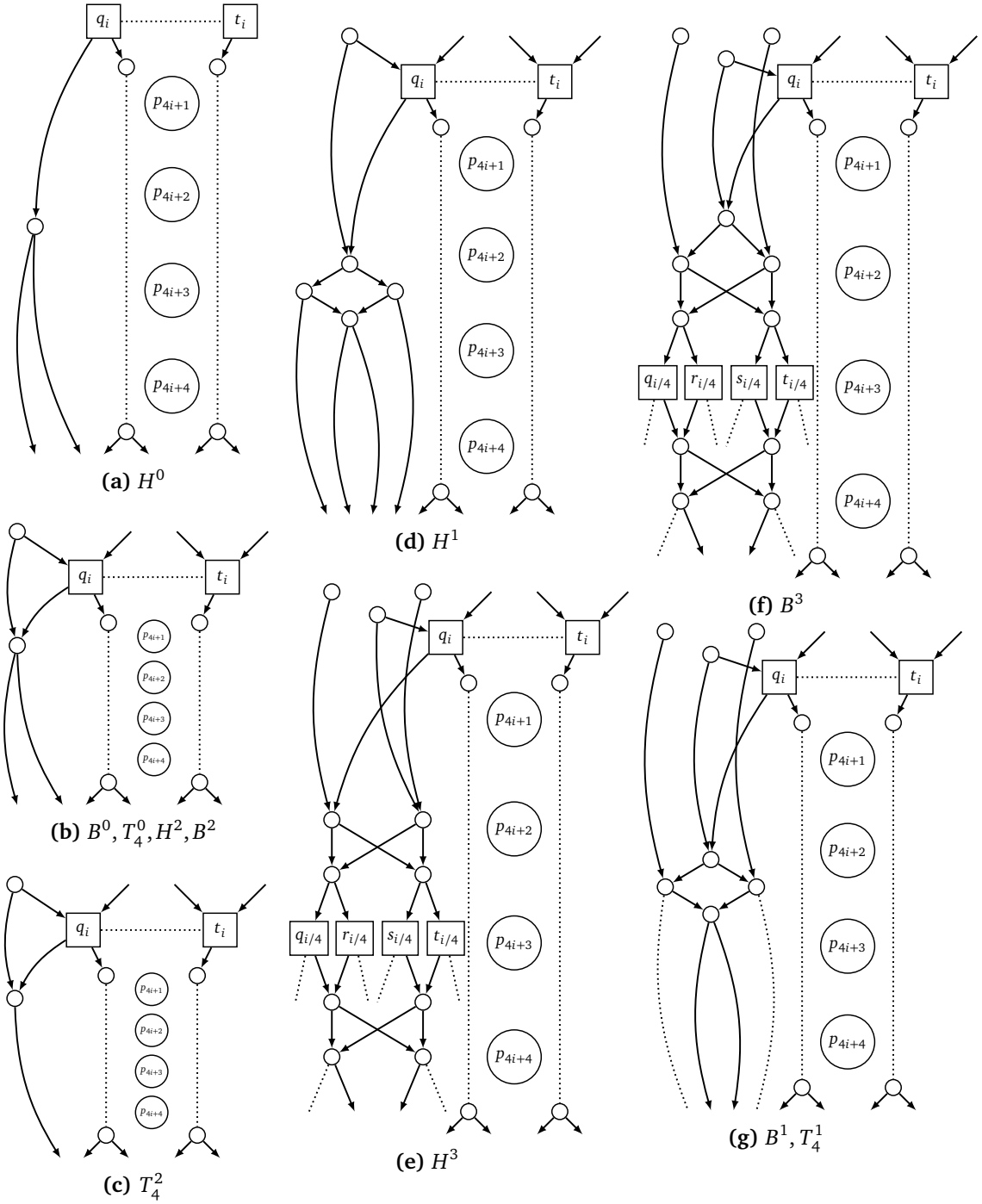
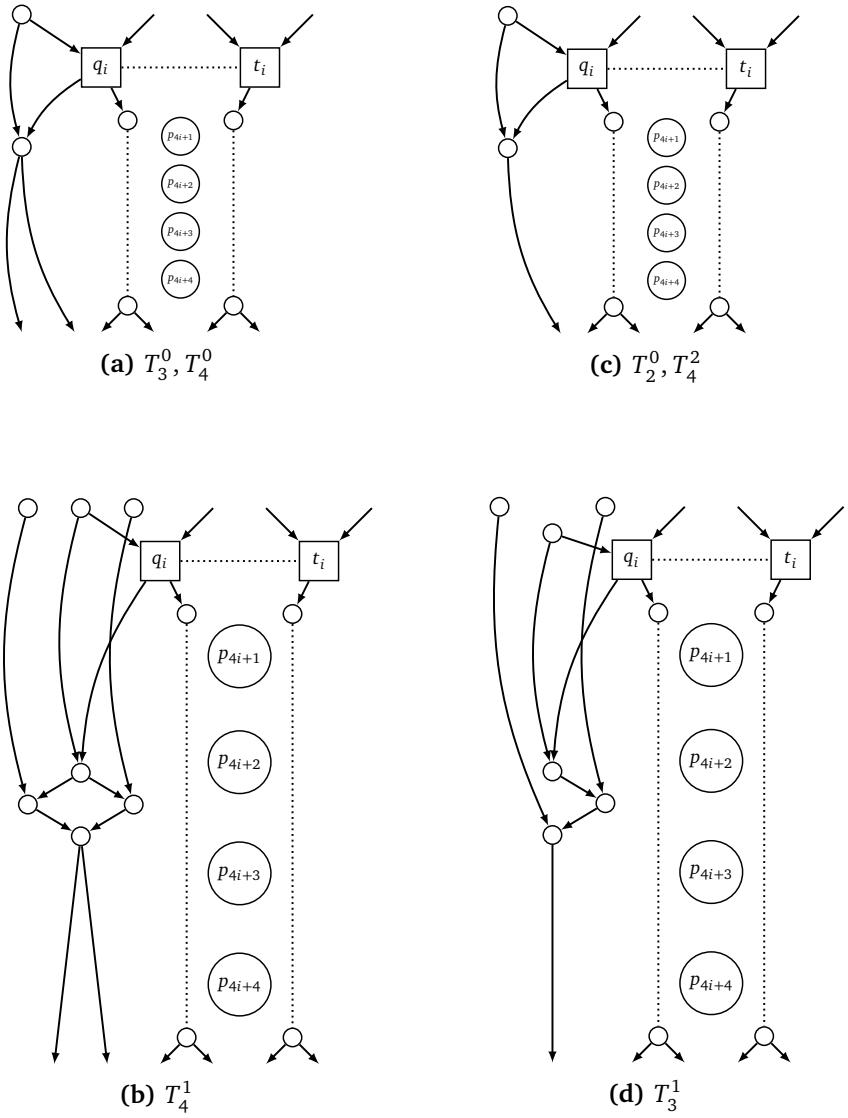


Figure 4.3: (a) and (b) show the body block construction with one subgraph node. (a) is the case for the first normal block and (b) the second. (c) is the case for the recursion step tail block. (d) and (g) are for recursion step head and body blocks respectively. (e) and (f) are both recursive construction cases where further subgraphs are created.



(a) T_3^0, T_4^0 **(c)** T_2^0, T_4^2

(b) T_4^1 **(d)** T_3^1

Figure 4.4: Our per-block tail constructions. (a) shows the case for the first subgraph node of tail recursion step block with 3 or 4 poles. (b) is the case for T_2^0 and T_4^2. (c) is the case for the second recursion step subgraph nodes of the tail block with 4 poles. (d) is the case for T_3^1.

$j \in \{q, r, s, t\}$ and T is the set of subgraph nodes generated in the current body block for each of recursion points $j \in \{q, r, s, t\}$. For every $s \in S$, the size of s can be either 0, 1, 2 or 3 as shown in Figure 4.3. The size of $t \in T$ varies and depends on the per-block type constructed. According to our approach, each block is constructed, ordered topologically then finally necessary node information are written to a file.

For simplicity, we explain our per-block body generation with $s_q \in S$ and $t_q \in T$ for a given body block $B_{4,0}^{P(l)}$ (where l is the number of additional hidden inputs in the programmable block and 4 and 0 are the number of upper and lower recursion points respectively) although the same construction is done for s_r, s_s, s_t and t_r, t_s, t_t (and for the other Γ_1 EUG outer skeleton block). s_q is constructed in order to set and pass the necessary outgoing wires to q_i (from Figure 4.3) and the subgraph nodes t_q. Figure 4.3 shows the different types of head, tail (with four poles) and body blocks constructed in our block UC construction approach. For the sake of simplicity, we do not show the interconnection of the outer skeleton poles, that are the same as in Figure 4.1a. For each of these body blocks, we eliminate the lower recursion points, since we only want to include each recursion point once. We do this because the construction of the subgraph nodes does not depend on the lower recursion points and also, the lower recursion points are the entry points to the next body block and we only need to save the outgoing wires of the last nodes in the outer skeleton and the subgraph in order to construct the upper recursion points of the next body block. The per-block construction denoted as $B_{4,0}^{P(l)}$ therefore contains four upper recursion points, four poles and fifteen nodes in the outer skeleton, and also in addition, the constructed subgraph nodes. Figures 4.3a and 4.3c show block constructions with only one subgraph node, for the first and last body blocks (and also other recursion step subgraphs) respectively. In Figure 4.3a, s_q is an empty set because there are no subgraph nodes at the head block level. Figure 4.3b also shows a block construction with only one subgraph node in both s_q and t_q. This scenario applies to the second body block (and also other recursion step subgraphs). Figure 4.3d has four nodes and one node in t_q and s_q respectively. Figures 4.3e and 4.3f are the two cases where our construction uses a recursion. They both have three nodes in s_q. These nodes, in particular, are from two different blocks $B_{4,0\{i-1\}}^{P(l)}$ and $B_{4,0\{i-2\}}^{P(l)}$ (where i is the current block number) with one node from the former and two nodes from the latter. The per-body block is recursively constructed using constructions depicted in Figures 4.3a-4.3g depending on the size of the DAG n.

Firstly, the outer skeleton body block is constructed, followed by the requisite subgraph nodes. All requisite subgraph nodes are constructed per-body block. To fully realize our block construction, two separate blocks (from left and right Γ_1 EUG) are merged by their poles, we term this combined blocks a joint block. The left and right outgoing wires of the poles are directed to the left and right nodes of the constructed blocks respectively. This applies to all block types (head, body and tail). In [GKS17], the left and right subgraphs of $U_n(\Gamma_2)$ are constructed separately. This is done to ensure proper wiring of universal gates. We explain this further later in this section and in Section 4.2.2.

Each joint block constructed is immediately destructed after writing the requisite information of the nodes to file. This is done to save memory by never storing $O(n \log n)$ information. Some requisite information are saved in order to correctly construct the next joint block. This information includes the outgoing wires of the last nodes in the outer skeleton and the subgraph and the running topological number. The outgoing wires of the last nodes are saved so that the next block's upper recursion points and subgraph nodes can be rightly set with input wires. The description of all nodes are written to a special file (one outer skeleton or different subgraph files). A node's description describes its gate or switch type and its incoming and outgoing wires. Once a joint block is constructed, the input wires to upper recursion points and the children subgraph nodes of the previous block's subgraph nodes are retrieved from file and set accordingly. In addition, the pole types are set depending on the current block number. The current block number being constructed is $\lceil \frac{pi}{4} \rceil$ where pi is the pole index. u input poles are set first, followed by k gate poles, then finally v output poles.

Other Body Block Construction Methods

Before finalizing our per-block approach of UC construction, we recapitulate an approach that we started off with but did not result in a successful UC generation. This approach did not combine both left and right $U_n(\Gamma_1)$ EUG blocks as a joint block. Instead, we generated the graphs separately block by block, while ordering and writing node descriptions and variables to file. A complication arises when it comes to setting input wires of gate and output poles. Since all gate and output poles depend on outgoing wires from nodes in both left and right $U_n(\Gamma_1)$ (due to merging), the current block construction has to be halted in order to begin construction of the other $U_n(\Gamma_1)$ EUG until the other incoming wire required by the gate pole is set. This also means that the wires of the gate poles' children nodes cannot be set. For output poles it is not as complicated as with gate poles since output poles have no children nodes depending on its output wires (children of output poles are cleared). This defies the purpose of our per-block UC generation approach. For input poles, this does not pose a problem because input poles have their output wires set as $w_{o_1} = w_{o_2} \in \{1, ..., u\}$ where u is the number of inputs of the DAG. Input poles are also considered to have no input wires (and therefore have their input wires cleared) which is an optimization in [KS16; GKS17] that additionally enables these wires to be listed as the first u wires.

4.2.2 Per-Block Topological Ordering

In this section we discuss our per-block (recursive) topological ordering of the joint block.

We mentioned before that we merge the left and right Γ_1 per-block constructions to form a joint block. After each joint block is constructed, we order the nodes of the outer skeletons and subgraphs topologically. Topological ordering is a precursor to writing the nodes' descriptions to file. In the ordering process, all nodes are assigned a topological value.Our topological

ordering is based on the Depth-First Search (DFS) algorithm. In [GKS17], ordering is started from the first pole's children (from the left and right EUGs $U_n(\Gamma_1)$) and worked down greedily. We also introduce a bottom-up ordering approach for each block where we begin ordering from the last nodes of the block in Figure 4.3. One reason for our bottom-up ordering is because we need to write the wires of the last two nodes accordingly to file and later retrieve them in the construction of the next block and hence we need to know their right order so we can easily set the input wires of the upper recursion points (of the next block) right. From the per-block generations depicted in Figure 4.3, it is evident that recursion points have children nodes from both the subgraph and the outer skeleton. Employing a top-bottom (starting from top nodes to bottom nodes of the block) ordering approach would include all nodes (but we want to order outer skeleton nodes and subgraph nodes separately to avoid too many simultaneous file opening and closing). The subgraph nodes are created and ordered recursively. We discuss this further in Section 4.2.3. Hence, we go by a bottom-up approach where ordering is started from the last two nodes (in total four for the joint block) of the outer skeleton block.

Listing 4.1: Bottom-up topological ordering for our per-block UC generation

```
1  procedure topologicalOrdering (u,v,k,t_n,pi)
2    Let u,v,k be number of inputs, outputs and gates of DAG respectively
3    Let pi be index of first pole in the current block
4    Let size_sub be size of constructed subgraph nodes at current blocks (left and
       ↪ right blocks)
5    Let stack be the stack
6    Let t_n be the running topological number from previous blocks written to file
7    Let size_B be size of the joint blocks
8
9    Let O_b be the toplogical order vector, resize to size_B
10   P ← {p_{pi+1}, p_{pi+2}, p_{pi+3}, p_{pi+4}}
11   Let u_{p(n)} be the number of input poles in this block
12   u_{p(n)} ← 0 // initialize
13   for i from 1 to 4 do
14     set pole as either input, output or gate based on u,v,k, and pi
15     if P_i is input do
16       u_{p(n)} ← u_{p(n)} + 1
17       Mark P_i as visited
18       Set topological number of P_i to 4*pi+i
19       Insert pole in O_b at position i
20     end if
21   end for
22   Let n_{l_1}, n_{l_1-1} and n_{l_2}, n_{l_2-1} be the last nodes in the two outer skeleton blocks
23   stack push n_{l_1}; stack push n_{l_1-1}; stack push n_{l_2}; stack push n_{l_2-1}
24   Let found_node be Boolean for topologically visited node found
25   while stack not empty do
26     Let n_c be the current node taken from stack
27     n_c ← stack top // node on top of stack
28     if n_c is marked as visited do
29       found_node = false
30       Let N_p be the parents of n_c
31       for all n_p ∈ N_p do
32         if n_p not marked as visited do
```

```
33            stack push n_p
34            found_node = true
35            break
36          end if
37        end for
38        if not found_node do
39          Set topological number of n_c to t_n
40          Insert n_c in O_b at position u_{p(n)}
41          u_{p(n)} ← u_{p(n)} + 1
42          t_n ← t_n + 1
43          stack pop
44        end if
45      else
46        Mark n_c as visited
47        if n_c is not a recursion point do
48          Let N_p be the parents of n_c
49          for all n_p ∈ N_p do
50            if n_p not marked as visited do
51              stack push n_p
52              break
53            end if
54          end for
55        end if
56      end if
57    end while
58  end procedure
```

Listing 4.1 shows our bottom-up per-block topological ordering of the outer skeleton block. With this algorithm, the outer skeleton nodes can be ordered without dealing with the subgraph nodes. We order the outer skeleton block first because otherwise this would involve the opening and closing of $1 + n_{sf}$, where $n_{sf} = 2 \cdot \sum_{i=0}^{log_4 n} 4^i$ is the number of subgraphs in the Γ_2 EUG and n is the size of the original DAG. Recursively one can calculate n_{sf} using the function from Listing 4.2. Our algorithm gets as input the number of inputs, outputs and gates of the DAG and also the last running topological number which was set and written to file during construction and ordering of the previously created block. This variable helps to continue the topological ordering from where we left off, without having to keep the whole structure in memory. Our next task is to resize the output vector to store the ordered nodes as seen in lines 8 and 9. This differs in the implementation of [GKS17] because they resize the vector with the size of the $U_n(\Gamma_2)$ EUG. We resize it to the sum of the two outer skeleton blocks and their subgraphs. This is followed by the setting of the poles of the joint block as either input, output or gates depending on the pole index and the values of u, v and k. In [GKS17], this is a prerequisite step to topological ordering as all pole types are set before ordering of the left and right Γ_1 EUGs. Thereafter, if any of the four poles in the outer skeleton blocks are input poles, we set its topological number and add it to the set of ordered nodes as seen in lines 15 to 20. We also set them as topologically visited. This flag helps to skip and identify ordered nodes during the ordering process as we iterate through nodes in the blocks.

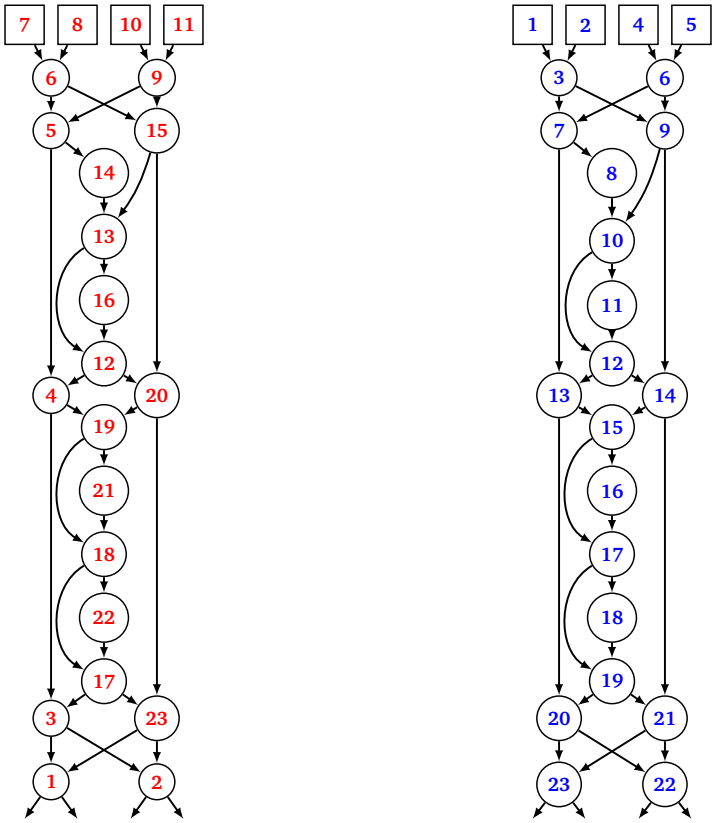
For the first step of the actual topological ordering, we put the last bottom two nodes of each block (in total four nodes) onto the stack (as seen in line 23). In [GKS17]'s implementation, the left and right children of the first pole in the head block are placed unto the stack. Now for every node on top of the stack, we check if it has been flagged as topologically visited. If not, we mark the node as visited then we check if any of its parents is also flagged as topologically visited. If any of its parent is flagged as visited, we push the first found parent node unto the stack and break from the loop (lines 46 - 52). Otherwise if none of its parents is flagged as topologically visited, we check the next node on top of the stack and repeat the process by checking if it is flagged as topologically visited. We skip unflagged nodes which are recursion points since there are no further nodes to check (no parent nodes). Now if a node on top of the stack is already flagged as visited, we still check if any of its parents is flagged as visited and push them onto the stack. Otherwise, we set its topological number and insert it into the set of ordered blocks as shown in lines 39 and 40. The index of the set of ordered nodes and the running topological number are both incremented by one (lines 41 and 42). The index is first initialized in line 12. In [GKS17], the DFS (topological ordering) is such that the last node in the U_n EUG is inserted first into the set of ordered blocks. Hence, the $topValue$ (index) variable is initially set to the maximum size of the EUG so that the last node is set in the last index. Afterwards, the index ($topValue$) gets decremented after every insertion into the ordered set. In our adaptation of the topological ordering, it's the reverse, starting from the bottom and working our way up (bottom-up ordering approach). Moreover, our approach works block by block, while the approach in [GKS17] handles the whole structure (that is stored in memory) at once. Therefore, we increment the index and topological number by one.

We show in Figure 4.5 that our bottom-up approach works effectively and results in a rightful topological ordering of the block. In Figure 4.5a, we show the order in which nodes are visited from bottom up. Nodes 1 and 2 are the last two nodes that we put first on the stack. With node 2 being on top of the stack, the algorithm greedily climbs up and checks its ancestor nodes. Figure 4.5b shows the resulting ordered nodes. It is noticeable that the right order is ensured.

4.2.3 Recursive Subgraph Generation

We describe our per-block recursive subgraph generation in this section.

Figures 4.3e and 4.3f show the recursive block constructions with Figures 4.3a, 4.3b, 4.3d, 4.3g and 4.3c being base cases. From Figure 4.3, each body block construction type is denoted either by H^n, B^n or T_x^n where $n = \{0, 1, 2, 3\}$ is the position of nodes between two poles in a head, tail or body in the subgraph and $x = \{1, 2, 3, 4\}$ denotes the type of tail block. A given subgraph has node(s) between every two set of recursion points of the parent graph to which this subgraph belongs. We know that the recursion points, for instance $\{q_1, ..., q_{\lceil \frac{n-4}{4} \rceil}\}$ are the poles of the next recursion step subgraph. n denotes the position of the nodes between two poles in a head, body or tail subgraph. We define H^n, B^n, T_x^n to denote constructions that can



(a) Order of visited nodes **(b)** Final topological order

Figure 4.5: Our bottom up topological ordering process. (a) shows the order in which nodes of the block are visited and pushed to the stack. (b) shows the topological order of nodes after bottom-up ordering.

Listing 4.2: Algorithm for calculating the number of recursion step subgraphs in $U_n(\Gamma_2)$.

```
1 procedure calculateSteps (n, c=0)
2   if n≤4 do
3     return 8·c
4   end if
5   s ← ⌈(n−4)/4⌉
6   c ← c+1
7   calculateSteps(s, c)
8 end procedure
```

be used to construct head, body and tail subgraphs respectively. For instance, Figure 4.3a can be used to construct only head subgraphs at the first position in a recursion step subgraph and is denoted as H^0. Figure 4.3b can be used in more than one case: B^0, T_4^0, H^2 and B^2. With our algorithm in Listing 4.4, we can generate the subgraph nodes recursively for all recursion steps at a given position for nodes n. Let us consider a simple example of a DAG with $n = u + k + v = 28$. Ideally, our approach constructs two of the same block from the left and right $U_n(\Gamma_1)$ EUGs for each of $\lceil \frac{n-4}{4} \rceil = 6$ blocks. In this scenario for $U_n(\Gamma_1)$, we have one head block $B_{0,0}^{P(l)}$ (since our per-block generation does not consider lower recursion points for a given block) which is then built up of H^0, H^1, H^2 and H^3 in order, five body blocks $B_{4,0}^{P(l)}$ which are all built up of B^0, B^1, B^2 and B^3 in order and one tail block $B_{4,0}^{P(l)}$ with 4 nodes, which is then built up of T_4^0, T_4^1 and T_4^2 in order. Constructing the first joint head block is straightforward as we do not have to construct any subgraph. We define the current subgraph level as $s_i = b_i - 1$ where b_i is the number of the current block under construction. There are $s_n = \lceil \frac{n-4}{4} \rceil - 1$ subgraph levels (which is equivalent to the total number of body blocks to be constructed per $U_n(\Gamma_1)$ EUG) in total for a given U_n. For $n = 28$, we have 5 subgraph levels. For our first three subgraph levels, we construct blocks with subgraphs as shown in Figures 4.3a (H^0), 4.3d (H^1) and 4.3b (H^2) respectively. At the fourth subgraph level, we construct our recursive case shown in Figure 4.3e (H^3). For the fifth and final subgraph level we construct the subgraph (together with the outer skeleton block) shown in Figure 4.4c (T_2^0). For this scenario with $n = 28$, we do not create further subgraphs from the recursion points in the subgraph (in fourth subgraph level). That is, there is only one set of recursion steps (one from each of eight recursion points from two blocks) generated from the recursion step graph (outer skeleton).

As mentioned before, tail blocks have either 1, 2, 3 or 4 poles. We show in Figure 4.4 the different tail subgraph constructions. Figure 4.4a shows the case for the first position of tail subraphs with 3 or 4 poles. Figure 4.4d shows a construction that can be used at the second position for tail subgraphs with 3 poles. There does not exist T_1^n (for tail blocks with one pole) because the construction of the nodes before the pole of the tail subgraph is already done by the head or body ($\{H_x^n, B_x^n\}$) constructions preceding, in particular our recursive constructions from Figures 4.3e and 4.3f.

The number of recursion step subgraphs in $U_n(\Gamma_2)$ can be calculated from Listing 4.2. The function is initially called with arguments n and count c as 0, where n is the size of the DAG. The function recursively calls itself and updates the subgraph level value and recursion step count until the number of remaining subgraph levels is 4 or less (base case), where we return $8 \cdot c$ (8 subgraphs in Γ_2 EUG). This function aids in the recursive subgraph generation as we will discuss further in this section. Figure 4.3f and 4.3g are applied in recursive cases with the number of subgraph levels greater than seven. That is, with $n \geq 40$. Figures 4.3e and 4.3f have variations where either one or both of the last two subgraph nodes have one or two outgoing wires. This is relevant to recursion steps with tail blocks.

Listing 4.3 shows the function that calls the recursive function in Listing 4.4. Listing 4.3 is called from the outer skeleton block and has as arguments the set of eight recursion points for the joint block (left and right), the number of poles in the subgraph and the position of the current block subgraph being constructed. The *recursiveSubgraphGeneration* function is called for each recursion point from both blocks as seen in lines 7 and 11. Listing 4.4 shows our recursive algorithm for generating subgraph nodes at a given subgraph level (see beginning of this section). For the base case of the recursive function, a recursion base subgraph node is created if the number of subgraph poles is less than or equal to 4 as seen in lines 5 and 6. We update the value of the number of subgraph poles in the current recursion step in line 10. For the generation of H^n, we check if the current subgraph level is less than or equal to 4 (line 11). We also create recursion points and further H^ns in the next subgraphs if we are at H^3 (lines 13 - 16). For the generation of B^n we firstly check if the remaining subgraph poles are more than or equal to 4 (lines 19 and 20). We do this because, with less than 4 poles, we can only create a tail recursion step block T_k^n (in the recursion step subgraph). We create new recursion points if we are at B^3. However, here we update the value of the subgraph level in the next recursion step (since H^0 can only be created at subgraph level 1 for a given recursion point) as seen in line 23. Finally, we construct T_x^n as seen in lines 30 to 35.

Listing 4.3: Function to call recursive function from outer skeleton

```
1  procedure callRecursion (rpts,s_p,c_l)
2    Let rpts be the upper recursion points vector of current block
3    Let s_p be the number of subgraph poles (recursion points) in a given recursion
       ↪ step
4    Let c_l be the current subgraph level/layer.
5    // left block
6    for i from 1 to 4 do
7      call recursiveSubgraphGeneration(rpts_i, s_p, c_l)
8    end for
9    // right block
10   for i from 5 to 8 do
11     call recursiveSubgraphGeneration(rpts_i, s_p, c_l)
12   end for
13 end procedure
```

Listing 4.4: Recursive algorithm for generating subgraph nodes

```
1  procedure recursiveSubgraphGeneration (r_p, s_p, c_l)
2    Let r_p be the recursion point
3    Let s_p be the number of subgraph poles (recursion points) in a given recursion
       ↪ step
4    Let c_l be the current subgraph level/layer.
5    if s_p ≤ 4 do
6      create recursion base subgraph node at c_l
7    else
8      rem_sp ← s_p − c_l − 1
9      Let n_s be the number of poles in the next recursion step subgraph
10     n_s ← ⌈(s_p − 4)/4⌉ − 1
11     if c_l ≤ 4 do
12       create head subgraph node(s) at H^(c_l mod 4)
13       if c_l mod 4 = 3 do
14         create new recursion points rpts_n
15         for i from 1 to 4 do
16           call recursiveSubgraphGeneration(r_p, n_s, 1)
17         end for
18       end if
19     else if rem_sp >= 4 do
20       create body subgraph node(s) at B^(c_l mod 4)
21       if c_l mod 4 = 3 do
22         create new recursion points rpts_n
23         n_l ← (n_s − 4)/4
24         for i from 1 to 4 do
25           call recursiveSubgraphGeneration(rpts_i, n_s, n_l)
26         end for
27       end if
28     else
29       Let x be the number of poles in the tail (subgraph) block
30       if rem_sp mod 4 = 0 do
31         x ← 4
32       else
33         x ← rem_sp mod 4
34       end if
35       create tail subgraph nodes(s) at T_x^(c_l mod 4)
36     end if
37   end if
38 end procedure
```

With this, we have shown how to generate topologically ordered Universal Circuits using the file system and achieve a scalable algorithm for Universal Circuit generation that stores at most $O(n)$ information in memory.

4.3 Universal Circuit Programming

In this section, we begin by discussing the stages in edge-embedding for Valiant's 4-way UC construction in Section 4.3.1. Finally, we discuss our per subgraph edge-embedding in Section 4.3.2.

4.3.1 Valiant's Edge-Universal Graph Edge-Embedding

We discussed briefly Valiant's edge-embedding algorithm in Section 3.3.1. Here, we discuss the stages in edge-embedding in Valiant's 4-way split UC construction as a primer to our per subgraph edge-embedding process.

After the UC generation, we do an edge-embedding of the circuit $C^k_{u,v}$ the UC represents to program the UC. There are three stages in the edge-embedding process: block edge-embedding, recursion point edge-embedding and the combination of both processes in a final step, as proposed in [GKS17]. We discuss the stages in the following sections.

Block Edge-Embedding

Each block is responsible for performing its own block edge-embedding, that is, programming the block nodes. According to Valiant, the underlying idea behind block edge-embedding is to map and pass the inputs of a block (upper recursion points) to poles, and also to map and pass poles to outputs (lower recursion points) [Val76]. [GKS17] formalizes the rules for the input and output vectors used for passing on inputs from upper recursion points to poles, and from poles to lower recursion points.

Recursion Point Edge-Embedding

Recursion point edge-embedding uses a process presented in [KS16] known as supergraph construction and is done at the subgraph level from the recursion points. Recursion point edge-embedding aims at programming the recursion point nodes and also to set the input and output vectors of the blocks for the block edge-embedding.

For Valiant's 4-way split UC construction, the supergraph construction generates four subgraphs. A supergraph is a graph $G_1 \in \Gamma_1(n)$ with a binary tree of Γ_1 graphs of decreasing size which define the embedding of every edge $e \in E$ from $G = (V, E)$ into U_n [KS16]. A given graph $G \in \Gamma_2(n)$ is split into two graphs $G_1 \in \Gamma_1(n)$ and $G_2 \in \Gamma_1(n)$. The splitting is done using the method of 2-coloring as discussed in Section 3.3. The second step is to merge each of both $G_1 \in \Gamma_1(n)$ and $G_2 \in \Gamma_1(n)$ into one $G_1 \in \Gamma_2(\frac{n}{2})$ and $G_2 \in \Gamma_2(\frac{n}{2})$ graphs respectively, by merging each two neighboring nodes of the graph to form one node in the merged graph. The merging is done to preserve node information [KS16]. Since we want to obtain four Γ_1 graphs, we repeat the process again by doing another 2-coloring then merging to obtain four Γ_1 graphs. This is equivalent to doing one 4-coloring and merging 4 nodes into one node [GKS17]. The entire process is repeated by merging each of the four Γ_1 graphs to form Γ_2 graphs then splitting to form Γ_1 graphs (both done twice) until the resulting Γ_1 graph has four or less nodes, that is, the recursion base is reached [KS16].

For every block, Günther et al. introduce input and output vectors that help to map edges in the DAG to paths in the block (for edge-embedding) [GKS17]. Head and tail blocks

are exceptions as they do not need input and output vectors, respectively. The input and output vectors are of x and y bits, respectively and they are defined based on the supergraph construction and they do not depend on the structure of the UC. For a k-way ($k = 4$ in our case), the input vector x has a value range of $\{0, ..., k\}^4$ and the output vector y has a range of $\{0, ..., 2k-1\}^4$ [LMS16; GKS17]. The output vector has a higher range so that a pole can be forwarded to either another pole or lower recursion point. Values $\{k, ..., 2k-1\}$ of the output vector are reserved for the recursion points.

Combined Edge-Embedding

The two edge-embedding methods are combined by first doing a recursion edge-embedding followed by a block edge-embedding. For each of the four subgraphs of $G \in \Gamma_1$ graph, we take each edge $e \in E$ we mark and set the control bit of the recursion points and set the input and output vectors of the corresponding blocks (where the edge can be found). At this point, the input and output vectors of all blocks have been set and hence the block edge-embedding is done. The process is repeated for all Γ_1 graphs of the supergraph and the recursion step graphs of the EUG (until the last Γ_1 graph of the supergraph with 4 or less nodes).

4.3.2 Per-Subgraph Edge-Embedding

From the Γ_2 graph, we embed each edge of the DAG as a path between poles of the Γ_2 EUG. For each edge $e \in E \subset VxV$ in the Γ_1 DAG $G = (V, E)$ (from the supergraph construction) that we want to embed, we need to set the input and output vectors first by doing a recursion edge-embedding.

We detail our per-subgraph recursion edge-embedding in Listing 4.6. The function takes as arguments the universal graph (subgraph) and its matching Γ_1 graph from the supergraph construction. Each recursion step subgraph information is stored in a separate subgraph file. We name the subgraph files such that, the name matches the position of the graph (subgraph) in the EUG and thus it is easy to select both the Γ_1 graph from the supergraph and its corresponding universal graph (subgraph). We perform the recursion point edge-embedding without going through the generated subgraph files first. We use one dedicated file for the Γ_1 graph and delete its content after successful edge-embedding of the graph. For the content of the file for the Γ_1 graph (of the supergraph), we write the input and output vectors of the Γ_1 graph of the supergraph, that is, we just read the Γ_1 graph in the supergraph and set all recursion point programmings as seen in lines 15 to 34.

For every edge $e = (v_i, v_j) \in E$ in the current Γ_1 graph, we locate the blocks in the Γ_1 EUG graph that contain both nodes v_i, v_j (lines 9 to 12). If both nodes are contained in the same block, the edge-embedding can be done in the same block so we set the control bit of the output vector at the corresponding node position (lines 15 to 17). Otherwise if v_i and v_j are not in the same block, we have to look for an edge $e' = (v'_i, v'_j) \in E'$ in one of the Γ_1 graphs

of the supergraph denoted as S_x (where x is the number of the graph in S as seen in line 22) and mark it. We set the programming bit of the input recursion point at position x to 1. If the nodes between the edge e' are in subsequent blocks, that is $b_j - b_i = 1$, we set the programming bit of the recursion point at position x to 0 (lines 24 and 25). This means the nodes are found in subsequent blocks and therefore the path has to enter and leave the next recursion step graph at the same node [GKS17]. If $b_j - b_i > 1$, we set the programming bit of the recursion point at position x to 1 and therefore the path enters the next recursion step graph without leaving at the same node. The input and output vectors of the block are then set as seen in lines 30 and 31. The process is repeated for all other edges $e \in E$ of the Γ_1 graph of the supergraph. We then write the input and output vectors in the dedicated file for the Γ_1 graph of the supergraph as seen in line 34. Now, all input and output vectors of the graph have been set and therefore we edge-embed all blocks (block edge-embedding) in graph G as seen in line 36 with the function from Listing 4.5. For the four subgraphs of G and the four Γ_1 subgraphs of G_1, we repeat the recursion edge-embedding (lines 37 - 39). We do this recursively until the recursion base cases are reached.

Listing 4.5 shows the block edge-embedding algorithm as formalized in [GKS17]. As mentioned before, in block edge-embedding each block takes care of its own embedding. The function takes as arguments the graph (subgraph) and the file containing the set of all input and output vectors of blocks in the Γ_1 graph. The graph's description and information is detailed in a file as we discussed before in Section 4.2. We know the number of lines that make up nodes of a head, body or tail block and therefore it is easy to get information of nodes in a given block. For each block and its corresponding input and output vectors, we set the control bits of the paths from the input (upper) recursion points to poles, between poles or the paths from poles to the output (lower) recursion points. For every input vector value, we set the path from the input recursion point to the corresponding pole, from pole to pole and from pole to output recursion points (lines 34 to 42 for body blocks). Lines 15 and 23 begin the procedure for head and tail blocks, respectively. Input and output recursion points are not considered for head and tail blocks, respectively.

Listing 4.5: Per-block edge-embedding

```
1  procedure blockEdgeEmbedding (G, f_io)
2    Let f_io be the dedicated file with the input and output vectors
3    Let n_b be the number of blocks in the Γ_1 graph
4    Let In,Out be the set of input and output vectors in f_io
5    In ← {in_1,...,in_{n_b}} // read from f_io
6    Out ← {out_1,...,out_{n_b}} // read from f_io
7    for i from 1 to n_b do
8      Let in and out be the input and output vectors of the current block B_i
         ↪ respectively
9      Let rp_0 and rp_1 be the four upper and lower recursion points of B_i
         ↪ respectively
10     Let P be the set of poles in B_i
11     in ← In_i
12     out ← Out_i
13     P ← {p_1,p_2,p_3,p_4}
14     if i = 1 do
```

```
15        for i from 1 to 4 do
16          if out_i ≠ 0 and out_i < 4 do
17            set path from P_i to P_{1+out_i}
18          end if
19          if out_i > 3 do
20            set path from P_i to rp_i^{i-3}
21          end if
22        end for
23      else if i = n_b do
24        for i from 1 to 4 do
25          if in_i ≠ 0 do
26            set path from rp_0^i to P_{in_i}
27          end if
28          if out_i ≠ 0 and out_i < 4 do
29            set path from P_i to P_{1+out_i}
30          end if
31        end for
32      else
33        //set paths
34        for i from 1 to 4 do
35          if in_i ≠ 0 do
36            set path from rp_0^i to P_{in_i}
37          end if
38          if out_i ≠ 0 and out_i < 4 do
39            set path from P_i to P_{1+out_i}
40          end if
41          if out_i > 3 do
42            set path from P_i to rp_i^{i-3}
43          end if
44        end for
45      end if
46    end for
47  end procedure
```

So far we described the edge-embedding process with one Γ_1 graph of the supergraph. We do the same with the other Γ_1 graph (and its subgraphs recursively) of the supergraph.

We reiterate that the above edge-embedding is made easy due to our per-block UC generation. The files are generated in such a manner identical to the Γ_1 graphs of the supergraph. That is, for every Γ_1 graph of the supergraph, there is a corresponding file from the subgraph of the Γ_1 EUG at the same graph position. So for every Γ_1 graph of the supergraph that has to be edge-embedded, there is a file containing the nodes' information of the corresponding Γ_1 EUG (subgraph). After edge-embedding, we have a resulting Γ_1 EUG file, programming file pair. That is, for every file (subgraph and outer skeleton files) created during the UC generation, there is a corresponding programming file.

With this, we have shown how to edge-embed the DAG using the scalable generated UC (in multiple files) and the Γ_1 graphs from the supergraph and achieve scalable per-subgraph edge-embedding.

Listing 4.6: Per-Subgraph recursive edge-embedding

```
1  procedure recursionEdgeEmbedding (G, G_1 = (V,E))
2    Let G be the subgraph (file)
3    Let R be the 4 recursion step subgraph files of G
4    Let G_1 be the Γ_1 graph in the supergraph
5    Let S be the 4 Γ_1 subgraphs of G_1
6    Let B be the set of blocks in G
7
8    for e = (v_i, v_j) ∈ E do
9      Locate (v_i, v_j) in file G
10     Let i' and j' denote the positions of v_i and v_j in their blocks
11     b_i ← ⌈i/4⌉
12     b_j ← ⌈j/4⌉
13     Let out [r_1] denote the output vector [recursion points] of B_{b_i}
14     Let in [r_0] denote the the input vector [recursion points] of B_{b_j}
15     if b_i = b_j do
16       if v_i ≠ v_j do
17         out_{i'} ← j' − 1
18       end if
19     else
20       Let s = (V', E') ∈ S denote the Γ_1 graph with e' = (p_{b_i}, p_{b_{j−1}}) ∈ E' and e' is not
             ↪ marked
21       Mark e'
22       Let x denote the number with s = S_x
23       Set the control bit of r_0^x to 1
24       if b_j = b_i + 1 do
25         y ← 0
26       else
27         y ← 1
28       end if
29       Set the control bit of r_1^x to y
30       out_{i'} ← x + 4
31       in_x ← j'
32     end if
33   end for
34   write input(in) and output(out) vectors to dedicated Γ_1 graph file f_io
35   // Edge–embed all blocks in G with all in and out vectors set so far
36   call blockEdgeEmbedding (G, f_io)
37   for i from 1 to 4 do
38     if S_i exists do
39       call recursionEdgeEmbedding(R_i, S_i)
40     end if
41   end for
42 end procedure
```

5 Implementation

In this chapter we discuss our implementation. We begin by detailing our scalable UC generation in Section 5.1. We finally discuss how our UC generation can be leveraged to program the UC in Section 5.2.

5.1 Scalable Universal Circuit Generation

In this section, we discuss the compilation process and development environment in Section 5.1.1. We then detail our per-block 4-way UC generation in Section 5.1.2. Thereafter, we discuss our scalable implementation using the file system in Section 5.1.3. We finally describe the functions of the various classes in our implementation in Section 5.1.4.
.

5.1.1 Compilation and Program Output

In this section, we discuss the tools and environmental setup we used for our implementation.

The Development Environment. Our per-block 4-way UC generation implementation was developed in C++ on Linux, specifically Ubuntu 16.04 Long Term Support (LTS). We used CLion as the Integrated Development Environment (IDE) which is a full-fledged environment which comes with a debugger, console and terminal to observe program output and the option to set and use a compiler of choice.

The Compilation Process. We used an open-source, cross-platform build tool known as CMake for building our program. CMake is also used to test and package software. A binary file is generated which can then be run. CLion comes with CMake pre-installed. The version of CMake we used is 3.7.

Program Output. The running time of the built program (binary file) varies and depends on the size n of the DAG, the higher the value of n the longer the running time, since it depends on the size of the UC $O(n \log n)$. We elaborate our evaluation in Chapter 6. After successful running, a number of output text files are created, the number of which also depends on n, more concretely, is linear in n, and equals to the total number of subgraphs of the UC plus one (outer skeleton file). The output text files contain the node descriptions of the outer skeleton and subgraph files (generated UC). The UC format is the same as that of [KS16]. Outer skeleton nodes are written to one file and subgraph nodes are written to different subgraph files (which enables the straightforward edge-embedding). There are also dedicated files that store information from previous generated subgraph nodes for later use. The content, number and size of the text files vary and we discuss this further in Section 5.1.3 and also in our evaluation in Chapter 6.

5.1.2 Per-Block 4-way Construction

We discuss implementation details of our per-block 4-way UC construction in this section.

Outer Skeleton Block Generation

We begin by defining the value of the size of the DAG. As mentioned before, for a given DAG of size n, we construct one head, one tail and $\lceil \frac{n-4}{4} \rceil - 1$ body blocks for the outer skeleton. There are a total of $\lceil \frac{n}{4} \rceil$ blocks to be constructed. We write the input wires $\{0, ..., u-1\}$ of input poles to the outer skeleton file, where u is the number of inputs of the DAG.

We mentioned in Section 4.2.2 the notion of a joint block as a merged block of the left and right Γ_1 EUGs at their poles in the outer skeleton. The joint head block is created first without upper recursion points (cf. Section 4.1). From our per-block UC generation, we only construct upper recursion points per-block of which the head block has none. Therefore we generate the joint head block without the upper recursion points. We begin the construction of the joint head block by creating a unique set of four poles $\{p_i, p_{i+1}, p_{i+2}, p_{i+3}\}$ (where i is the pole index of the current joint block) for the joint block (both head blocks) as shown in Figure 4.1f. For the joint block $i = 0$ therefore we generate poles $\{p_0, p_1, p_2, p_3\}$. We then create all nodes for both head blocks in the joint block and set the children and parents of each node. We borrow the construction of nodes from [GKS17]'s implementation.

We do a topological ordering of each constructed joint block. We discuss this further later in this section. For each constructed joint block, we write to file the outgoing wires of the last nodes in the left and right blocks (of the joint block). The last nodes are the nodes with outgoing wires that are the input wires of the lower recursion points as seen in Figures 4.1f

and 4.1a. We write these outgoing wires so that we can set the input wires of the upper recursion points of the next joint block construction[1].

As shown in Figures 4.3b to 4.3f it is evident that the left input wires of upper recursion points also have to be set and we obtain this from our per-block subgraph generation as we will discuss later in this section. We do not save last wires of tail blocks as they end our per-block UC generation.

The same procedure for constructing the head blocks is applied to the construction of body and tail blocks. Tail blocks however have $\{p_i, ..., p_m\}$ poles created where $m = n \mod 4$. For each joint body and tail block, in addition to the created poles and nodes, we create upper recursion points for both left and right blocks. We read from a special dedicated file the output wires of the last nodes written. We set these as the input wires of the block's upper recursion points accordingly. Also, if any of the created poles is an output pole, we write its output wire to file. We update the output poles' wires for every joint block and only write it (to the outer skeleton file) lastly after generating and writing the tail block nodes. The descriptions of all nodes and poles in the joint block are written to the outer skeleton file. For every node in the set of ordered nodes in the joint block, if the node has 2 parents and is not an input pole, we write its node description to file. There is the special case of upper recursion points which have no parent nodes as they are the top nodes in our per-block generation. Since we set the input wires of the upper recursion points during the block construction (body and tail blocks), we count the size of the input wires set and write the description accordingly. For the first joint body blocks, we set only one input wire and hence the upper recursion points are reverse Y switches that can be replaced by wires only as described in [KS16]. This is the case shown in Figure 4.3a.

As mentioned before, a node description indicates the type of the node and its input and output wires. We encapsulate this information in a string. To obtain this string, we set the type of the node first. Types include X, Y or U where X is an X switch, Y is a Y switch and U is a universal gate as described in [KS16]. The input and output wires of the node are concatenated to the type. A node having 2 parent nodes and 2 children nodes with input and output wires $\{227, 218\}$ and $\{230, 231\}$ respectively has its description as "X 227 218 230 231". For gate poles (with 2 input wires and 1 output wire (the same output wire is duplicated as two output wires)), we set the type as U because the same outgoing wire goes to both of its children nodes. For instance a gate pole's description may be denoted as "U 233 241 242" where 233 and 241 are the input wire numbers and 242 is the output wire number. A node's output wire(s) is the sum of its topological number and the additional wires. We discuss this further later under topological ordering in this section. For X switches the second output wire is the first output wire plus one which is evident in the above example. It is noticed that the left children of recursion points are not created in the joint block generation and hence the input wires of the right (second) children are set accordingly as output wire plus one. Y switches have two input wires and one output wire. An example is "Y 138 141 142".

[1] Alternatively, the outgoing wires of the last nodes can be stored in memory, since we need to store only 8 numbers per-block for the entire UC generation.

In addition to only joint body blocks, we calculate the value for the number of subgraph poles in the next recursion step $subPoles = \lceil \frac{n}{4} \rceil - 1$ and current subgraph generation level $c_l = \frac{pi}{4}$ (where pi is the pole index of the current joint block) as shown in our recursive subgraph nodes generation algorithm in Listing 4.4. The upper recursion points $rpts$ are already created from the outer skeleton generation (joint block). We then start generating the subgraph nodes recursively.

Finally, after generating all outer skeleton nodes and subgraph nodes, we write v output pole wires to file.

Topological Ordering

As mentioned before, we use a bottom-up topological ordering approach which we discussed in Section 4.2.2. Our bottom-up approach results in the upper recursion points amongst the top ordered nodes, hence a topological ordering of the DAG.

In addition, we set each created pole as either an input, gate or output pole. From $n = u+v+k$, there are u input poles, v output poles and k gates. For input poles, we also clear its parents, that is we remove association with its parent nodes. This is an optimization from [GKS17] which aids in faster writing of nodes' descriptions since we do not have to consider input poles. Also, this is done because most circuit description formats require the first u wires to be the input wires.

We described the purpose of the running topological number in Section 4.2.2. We begin the topological ordering process by reading the value of the running topological number. Likewise, we read the value of the number of additional wires set in previous block generations. The number of additional wires indicates the additional wires set in addition to the topological values of nodes (including those already set). We have variations of our implementation where we either store the running topological number and additional wires count in memory or file[2]. For head blocks, these values are set to 0.

After successful topological ordering of all joint block poles and nodes, we set the number of additional wires for each node. For an X switch, we have an extra wire accounted for by the last outgoing wire. A reverse Y switch has the same input wire duplicated as output wires which is an output wire from its parent node so we decrement the number of additional wires. A Y switch has only one outgoing wire and hence we neither increment or decrement the number of additional wires count. This idea is also borrowed from [GKS17]'s implementation. For each node, we can then calculate its outgoing wire as the sum of its running topological number and its additional wires count. The final values of the running topological number and the additional wires count are saved to memory.

Either the left or right block can be topologically ordered first without affecting the outcome of the UC generation. To ensure this uniformity, we must ensure the order of the last nodes

[2]For our experiments in Chapter 6, we store these values including the outgoing wires of the last nodes in memory making it a total of 10 numbers stored in memory per-block generation for the entire UC generation.

of either left or right blocks put on the stack (as seen in Listing 4.1) is maintained for each per-block generation of the outer skeleton. If the last nodes of the left block are put on the stack first, the right block is topologically ordered first. This is particular for blocks with only input poles. This does not apply to blocks with gate poles as gate poles require input wires from both left and right block nodes and can only be ordered if its parent nodes are already ordered according to the topological ordering.

Recursive Subgraph Generation

The generation of subgraph nodes is done recursively as shown in Listing 4.4. For every joint body block generation, we generate the subgraph nodes for both left and right blocks.

We generate a file for each recursion step subgraph and hence we need a way to detect the recursion step subgraph location of the nodes being generated. The nodes generated are written to a file named after this location. We begin this naming with "l" and "r" for the left and right Γ_1 EUGs, respectively. We know that $Q = \{q_i, ..., q_{\frac{n-4}{4}}\}$ are the poles of the first recursion step subgraph (and also for the three other recursion point sets R, S, T). Let c_l be the position of the subgraph nodes being generated which is situated between two subgraph poles q_i and q_{i+1}. For each recursion point, we append either of $\{q, r, s, t\}$ to the current value of the location. The process is repeated recursively for the naming of the location for every recursion step subgraph, continuing from the last set location name. For the first body block generation $c_l = 1$, we have "lq" as the only location of the subgraph nodes being generated belonging to the first recursion step subgraph (from recursion point q) of the left Γ_1 EUG. We therefore write the descriptions of the subgraph nodes to a text file named after this location (one line per node description), "$lq.txt$". For all other c_l values we write the descriptions of nodes to this same file for the first recursion step subgraph from point q. For $n \geq 49$ at a given $c_l = 9$, we write subgraph nodes for two recursion steps at locations $\{lq, lqq, lqr, lqs, lqt\}$ for nodes belonging to initial recursion step point q from the outer skeleton block.

We introduce tags for each subgraph node. A tag is a unique identification for subgraph nodes of a given recursion step subgraph at a given c_l. A node's tag is a concatenation of its subgraph location and its position, that is c_l. For instance, subgraph nodes in the first body block generation have tags $\{lq1, lr1, ls1, lt1, rq1, rr1, rs1, rt1\}$. With tags we can easily obtain the descriptions of nodes in the set of last subgraph nodes from previous blocks generated S as discussed in Section 4.2. This helps to reduce search times for descriptions of the last subgraph nodes of S, with a constant number of $O(1)$ lines read in file (only one line of string in every tag file), since we do not have to search through larger subgraph files for the right descriptions.

With our algorithm in Listing 4.4, we generate the subgraph nodes recursively for all recursion steps at a given c_l. For $H^{c_l}, B^{c_l}, T_x^{c_l}$ we generate subgraph nodes as seen in Figure 4.3. For each set of recursion step subgraph nodes generated at a given c_l for all recursion steps, we

create text files named after the tag of the subgraph nodes. We write the descriptions of the subgraph nodes as a string in one line to the file. This reduces the complexity of reading the written information. We do this to read the outgoing wires of the nodes in S when we need them. For Figures 4.3b to 4.3g we need to get the outgoing wires from the set of subgraph nodes S. Listing 5.1 shows how we do this in C++. In addition to passing the location and c_l as arguments to the function, we pass variables $dist$, $step$ and pos. $dist$ is the difference between the current c_l and the c_l of the last node(s) being searched for. $dist$ can either be 1 or 2. $step$ is a Boolean value to check if the last subgraph node(s) are in the next recursion step or not. $pos \in \{0,1,2,3\}$ is the recursion point position of the last subgraph node(s) which we require when searching for subgraph nodes in the next recursion step. In line 5, we initialize the tag as an empty string. We next check if we need to look for nodes from the next recursion step. If so, we update the location to search accordingly as seen in lines 7 to 16. For the next recursion step, c_l is updated to $\lfloor \frac{c_l - 4}{4} \rfloor$ as seen in line 15. A given location $lqq34$ with $pos = 2$, $step = true$ and $dist = 1$ will yield $tag = lqqs7$. If we do not have to search for subgraph nodes in the next recursion step, we set the tag of the subgraph nodes as the current location concatenated to the $c_l - dist$ as seen in line 18. We use the tag to open the right text file to and read the line as seen in lines 20 to 24. Finally, we put each description in the string into a vector and return it as seen in lines 27 to 34.

After obtaining the descriptions of the nodes in S, we set the input wires of the subgraph nodes generated (T) as seen in Figures 4.3b to 4.3g. Before setting the input wires, we generate all the nodes required in either H^{c_l}, B^{c_l} or $T_x^{c_l}$. Next, the subgraph nodes in T are ordered topologically. The number of nodes in T are either of $\{1,4,12,13\}$ (cf. Figure 4.3). These are relatively fewer nodes than the nodes in the outer skeleton and therefore we order the subgraph nodes manually. We also set the number of additional wires for each node. We then write the subgraph nodes descriptions to file as discussed. For every recursion point in the subgraph nodes, in particular H^3, B^3, we generate nodes in the next recursion step subgraph recursively. If the generated subgraph nodes are in the first recursion step subgraph, we write down the outgoing wires of the last node, in particular the right (second) outgoing wire. We do this because the wire is the first (left) input wire required by the upper recursion points in the next outer skeleton joint block construction.

Listing 5.1: Algorithm for obtaining descriptions of nodes in previous subgraph levels

```
1 std::vector<std::string> Subgraph::getPreviousSubgraphNodesDesc(
    ↪ std::string location, uint32_t currentLevel, int dist, bool
    ↪ step, int pos) {
2   Let pos be the recursion point position of the node wires being
      ↪ searched for
3   Let step be the flag for searching in the next recursion step or
      ↪  not
4   Let dist be the difference between the current subgraph nodes
      ↪ position and the position of the nodes being searched for
5   std::string tag = ""; //
```

```
6     // search for subgraph node wires from last block level in the
          ↪ next recursion step
7     if(step) {
8       switch(pos % 4) {
9         case 0: { location += "q"; } break;
10        case 1: { location += "r"; } break;
11        case 2: { location += "s"; } break;
12        case 3: { location += "t"; } break;
13        default: break;
14      }
15      int newLevel = (int) (currentLevel - 4) / 4;
16      tag = location + std::to_string(newLevel);
17    } else {
18      tag = location + std::to_string(currentLevel - dist);
19    }
20    std::ifstream stream (tag + ".txt");
21    std::string line = "";
22    while(getline(stream, line)) {
23      if(line != "") {
24        break;
25      }
26    }
27    std::vector<std::string> result;
28    std::istringstream iss(line);
29    for (std::string token; std::getline(iss, token, ','); )
30    {
31      result.push_back(std::move(token));
32    }
33
34    return result;
35  }
```

5.1.3 Scalable File Generation

From our per-block UC generation, each joint block and subgraph (at a given subgraph level) nodes descriptions are written to file. The result of our per-block UC generation is having the descriptions of nodes in files. The number of files created and written to is equivalent to the total number of subgraphs generated for the two Γ_1 EUGs plus the file created for the outer skeleton. We denote this number as $f_n = 1 + n_{sf}$, where n_{sf} is the result of the function in Listing 4.2 for a given $U_n(\Gamma_2)$ of size n. The files are generated according to the subgraph structure of the Γ_1 graph from the supergraph and this aids in easy edge-embedding of the graph into the EUG. This makes it easier to locate the subgraph (file) corresponding

to the Γ_1 graph from the supergraph which contains the information we need for successful edge-embedding as discussed in Section 4.3.2. In addition, we create so called tag files for both writing and reading of descriptions of subgraph nodes when we need information from previously generated subgraph nodes. This helps reduce search times for descriptions of the last generated subgraph nodes since we do not have to search through larger subgraph files for the right descriptions (cf. Section 5.1.2).

We encode our files using ANSI or UTF-8 as a character would be stored in one byte. We use sequential access when reading and writing files. We do this because our construction is based on a topological order of nodes and are written to file sequentially in this order. Likewise this is essential during the edge-embedding process as we traverse nodes (lines) in sequential order and hence we read files in the same manner.

As mentioned before, we create a text file for appending for each recursion point subgraph. We create the tag files for both reading and writing. We have variations of our implementation where we create these subgraph files (not tag files) beforehand or on demand for the first time. By default, we create the subgraph files dynamically for the first time when node(s) description(s) have to be written.

It is evident that as our per-block generation progresses, the number of tag files increases. Also, for a given set of generated subgraph nodes, we may require output wires of previous subgraph nodes from at most position $c_l - 2$. We therefore discard tag files for subgraph nodes at position $cl - 3$ for every generated subgraph nodes (at a given subgraph level (cf. Section 4.2.3)). At the end of the UC generation, we have only 3 tag files per recursion step subgraph (which we can then discard). Listing 5.2 shows how we delete unneeded tag files. Given a tag as argument (for example "lqqr7.txt"), we extract the letters and numbers in the tag as seen in lines 6 to 12. If the extracted number is less than 4, we return from the function (because we will need information from those files later), else we set the tag to be deleted and delete the file accordingly as seen in lines 16 to 22.

Listing 5.2: Deleting Unused Tag Files

```
1  void FileWriter::deleteUnusedFile(std::string tag) {
2    std::string numbers = "1234567890";
3    std::string letters = "";
4    uint32_t number = 0;
5    // extract letter
6    for(int i = 0; i < tag.size(); i++) {
7      if(numbers.find(tag[i]) == std::string::npos) {
8        letters += tag[i];
9      } else {
10       // then rest of tag is the number
11       number = stoi(tag.substr(i, tag.size() - i));
12       break;
13     }
14   }
```

```
15    // nothing to delete
16    if(number < 4) {
17      return;
18    } else {
19      uint32_t toDelete = number - 3;
20      std::string newTag = letters + std::to_string(toDelete);
21      std::string fileName = newTag + ".txt";
22      std::remove(fileName.c_str());
23    }
24  }
```

5.1.4 Program Description

We describe the classes used in our per-block 4-way UC construction in this section. The underlying implementation from [GKS17] defines classes UCNode and ValiantUC already, which we changed according to our needs.

ValiantUC. Class *ValiantUC* initiates our per-block 4-way UC construction (cf. Section 4.2). For a given DAG of size n with u inputs, v outputs and k gates, we set these values accordingly. We write u inputs to the outer skeleton file. The number of blocks to be generated in the outer skeleton is then computed (Section 4.2). We call our *Block* class iteratively to generate the blocks. Finally, v output wires are written to file.

Block. The *Block* class creates all joint head, body and tail blocks [GKS17], which consist of nodes of the class *UCNode* below. It also creates the edges between the nodes for the different types of blocks. In addition, it calls the *Subgraph* class to generate subgraph nodes per-block. The *Block* class is also responsible for the bottom-up topological ordering of the block nodes in the outer skeleton (cf. Sections 4.2.1 and 5.1.2). It also sets the output wires need for the right input wires of the upper recursion points in the next per-block generation (Sections 4.2 and 5.1.2).

Subgraph. Class *Subgraph* recursively generates all recursion point subgraph nodes and creates the edges between the nodes, per-block generation (Figure 4.3). For the fewer nodes generated, it also manually orders them (Section 5.1.2). Class *Subgraph* also sets the tag file subgraph nodes are written to as well as tag files from which to read (Section 5.1.2). It is also responsible for the setting of the last output wire to the next subgraph pole (recursion point of outer skeleton).

FileWriter. $FileWriter$ takes care of all file writings. It is called by $Block$ and $Subgraph$ to write descriptions of outer skeleton nodes and subgraph nodes respectively as discussed in Section 5.1.2. Wires of input and output poles are also written by $FileWriter$. It also writes subgraph nodes to their respective tag files.

FileReader. Analogous to $FileWriter$, $FileReader$ is responsible for all file reading operations.

UCNode. We inherit some of the functionality of $UCNode$ from [GKS17]'s 4-way split UC construction implementation. Class $UCNode$ creates all node types and functionalities associated with nodes such as setters and getters for input wires, description, children, parents, topological value and additional wires count.

MemStore. $MemStore$ stores the updated additional wires count and topological numbers. We have setters and getters for these variables.

5.2 Scalable Universal Circuit Programming

We leave programming of the UC as future work. In this section, we discuss how this can be implemented using our scalable UC and files generation solution.

As mentioned before in Section 4.3, we embed each edge $e \in E \subset VxV$ of the Γ_1 graph from the supergraph construction. We know from before that each block takes care of its own embedding. We can therefore do the block edge-embedding of the outer skeleton and subgraphs, using their respective generated files. For the block edge-embedding of the first left and right Γ_1 EUGs, we refer to the outer skeleton file for the nodes' descriptions. We know the number of nodes (lines) that constitute either a joint head, body or tail block. We then obtain the nodes descriptions corresponding to the left or right Γ_1 graph and set the control bits of the paths respectively using the input and output vectors set by the recursive edge-embedding. We write the control bits in a file P_x for programming bits of nodes in the outer skeleton, where x is the location of the graph. For any other subgraphs of the two Γ_1 EUGs, we refer to the corresponding subgraph files for the nodes' descriptions. We write the programming bits of the subgraph nodes into separate files as well, using a naming system similar to the subgraph files' names (for example, P_{lq}).

We discussed in Section 4.3 our per-subgraph recursion edge-embedding. For every Γ_1 graph of the supergraph, we do the recursive edge-embedding and write the input and output vectors to a file. We then do a block edge-embedding of the subgraph file (that matches the position of the Γ_1 graph in the supergraph) using the input and output vectors just set. For example, for the block edge-embedding of subgraph with nodes contained in file name $lq.txt$, we call the function in Listing 4.5 with the subgraph file ($lq.txt$) and the input and output

vectors just set for the Γ_1 graph matching this subgraph. Our scalable file generation makes this particularly convenient and very intuitive. We do this recursively until the recursion base case of the Γ_1 graph (of the supergraph) is reached.

We iterate that the edge-embedding is made easy due to our scalable file generation. Each Γ_1 graph of the supergraph has a corresponding subgraph file and hence it is easy to find the right blocks to do both recursive and block edge-embeddings.

6 Evaluation

In this chapter, we evaluate the performance of our per-block 4-way UC generation and also compare our results to that of [GKS17]'s UC generation. We begin by discussing our evaluation criteria in Section 6.1. Thereafter, we elaborate our process for obtaining the measurements and also explain the results of our experiments in Section 6.2. We show the improvements we made in memory usage and the price we paid in runtime and disk space usage, which are both acceptable since they are only linear in n.

6.1 Evaluation Criteria

In this section we discuss the metrics we use for our evaluation, more specifically, memory consumption in Section 6.1.1, necessary disk space in Section 6.1.2 and runtime of the algorithm in Section 6.1.3.

Environment: We run our experiments on an Ubuntu 16.04 server with 4 x 8 GB of usable DDR3 memory clocked at a maximum frequency of 2400 MHz and 1TB of SSD storage. The processor is an Intel Core i7-4790 at 3.6 GHz.

6.1.1 Memory Consumption

Memory consumption is a common metric used to test the performance of programs as this signifies the efficiency of the program, that is, its usage of resources. For high performance computing, this is particularly crucial as inefficiency in memory usage means a program may overwhelm its allocated memory and also can affect the program's runtime and output. We are particularly interested in memory consumption because the memory usage of previous implementations was very huge. For instance, we compare the memory consumption of our per-block UC generation to Günther et al.'s in Section 6.2.2.

We use the server machine's $proc$ file system to obtain information on the program's memory usage. The $proc$ file system is a file system on Unix machines that contains information about the system. Information about a running program or process that we can obtain include the process ID, the user running the program, the command with which the program was run, the CPU usage, the memory usage amongst others.

Resident Set Size (RSS) and Virtual memory size are two examples of memory information of the process (running program) that can be obtained. Virtual memory is all the memory to which the running program can have access. RSS is the real memory size allocated to the program which is currently existent in Random Access Memory (RAM). We are particularly interested in RSS as this includes heap and stack memory and excludes memory that is swapped out, that is data moved from real memory and stored on disk (virtual memory includes swapped memory) for later access.

6.1.2 Disk Space

In addition to memory consumption of the program, we check the program's usage of the file system for storage. As discussed before in Chapters 4 and 5, our UC generation relies on the file system for storage. We use more disk space to avoid memory overflow. This amounts to many files used but then most of them are deleted except the outer skeleton and subgraph files. In addition, for every subgraph, there remains 3 associated tag files as mentioned in Section 5.1.3. We are therefore concerned with how much of the file system storage our program uses.

6.1.3 Runtime

As mentioned before, we generate a lot of files in our implementation and this yields to a lot of opening, closing, reading and writing of files. We are therefore concerned about the runtime and how worse the algorithm becomes due to many IO operations. This is very important to measure as too high runtimes may contribute to time being a bottleneck.

6.2 Experiments and Results

In this section we discuss our evaluation results. We compare our benchmarks with the 4-way UC construction of [GKS17] because it is more scalable than Valiant's 2-way split construction [Val76]. Also, it has better UC sizes as shown in the results [GKS17]. They show that from $n = 212$ on, the 4-way split construction is almost always better than the 2-way split construction. Our implementation is based on Günther et al.'s 4-way construction because it is more modular and therefore easier to make scalable.

Our implementation uses only tail blocks with 4 poles as shown in Figure 4.1e and therefore we run the experiments for only certain numbers. With some more engineering effort, the remaining tail blocks with 1, 2 and 3 poles can be implemented [GKS17, Figure 4.1] as seen in Section 4.1.

In the remainder of this section we explain our results. We begin by discussing our process of obtaining our measurements for evaluation in Section 6.2.1. We then discuss the results of our experiments in Sections 6.2.2, 6.2.3 and 6.2.4 for memory consumption, runtime and disk space usage respectively. Finally we discuss the size of our UC and tabulate some of our results in Section 6.2.5.

6.2.1 Process of Obtaining Measurements

As mentioned before, we obtain our program's memory usage from the proc file system. We do this using the *ps* command which shows the status of processes. We gather this information using a bash script running as a background process. The bash script inspects the output of the *ps* command with our program passed as an argument. For each output returned by *ps*, we get the command with which we run the program to obtain the value of n. We then write the captured information of the program into a file named after the value of n. Captured information are the process ID, the percentage of RSS memory and the actual amount of RSS used by our program. This bash script runs the aforementioned steps in a $while$ loop until it is explicitly killed. We then run a python script to compute the sizes of the generated UC and subgraph files after program execution is done for a given n.

6.2.2 Maximum Memory Consumption

Figure 6.1 shows a memory benchmark graph of our per-block UC generation compared to [GKS17]'s generation. The figure shows a graph of maximum RSS memory used in MB with varying number of nodes n, of the original DAG for which the UC is created. It is evident that our per-block UC generation has a linear outline and for 2 446 676 nodes, the program consumes less than 2 GB of memory. This constitutes a maximum of only 4.7% of the 32 GB available memory on our benchmarking environment. We can therefore generate UCs with millions of nodes on devices with limited amount of usable memory, for instance 2 GB or 8 GB. It is also possible to run multiple UC generations in parallel on big machines, and even more importantly, we can now generate UCs of billions of nodes. From $n = 8$, our per-block generation grows slowly and steadily throughout until the last tested value for $n = 2\,446\,676$ is reached. [GKS17]'s generation rapidly rises from $n = 8$ and continues at this pace until $n = 1\,135\,956$. This represents 28 GB of memory used. From here, it rises again in memory consumption but at a lesser pace than before until the maximum available 32 GB of memory is used. From here, it uses up all the 32 GB of memory and then takes longer since it uses swapped memory to overcome the memory limitation which still works for $n = 1\,398\,100$ but not for $n = 2\,446\,676$. [GKS17]'s generation failed to complete run for $n = 2\,446\,676$ due to a bad_alloc fault (from using up all of the allocated memory) which is evident from the lack of coordinates for $n = 2\,446\,676$ from Figure 6.1.

[GKS17]'s implementation generates all nodes for the Γ_2 EUG and stores them in memory, hence is heavily dependent on memory. In contrast, our per-block approach generates nodes

on a per-block basis (and deallocates memory after each block) as discussed in Chapters 4 and 5 and therefore requires memory that is only linear in the size of the input circuit n. More precisely, it is linear in the number of subgraphs which is, on the other hand, linear in n. This linearity can be estimated as $2 \cdot \sum_{i=0}^{log_4 n} 4^i = 2 \cdot \frac{(4n-1)}{3}$.

This is justified by the graph that is linear in n, for our per-block UC generation ($O(n)$) and with a logarithmic ($O(n \log n)$) factor larger for [GKS17]. We therefore conclude that our per-block UC generation runs efficiently using far less memory than the implementation of [GKS17] and can run on memory-restricted devices with for instance 2 GB of memory for Boolean circuits with over 2 million nodes. For high performance computing using powerful servers, we can run multiple instances of our per-block UC generation for Boolean circuits with millions of nodes.

6.2.3 UC Generation Runtime

Figure 6.2 shows a graph of runtime in milliseconds with growing input graph sizes n of our per-block UC generation compared to the UC generation of [GKS17]. [GKS17] runtimes rise steadily until about 400 000 poles ($n = 349\,524$). There is then a small spike in the rate of increase until about 1 200 000 poles ($n = 1\,135\,956$). Hereafter, there is a slight surge in increment until $n = 1\,398\,100$ is reached at which point almost all 32 GB of memory is used but runs successfully due to the use of swap memory. Our per-block UC generation increases gradually with intermittent rises between n values at $n = 349\,524$, $n = 611\,668$, $n = 87\,380$, $n = 1\,135\,956$ and $n = 1\,398\,100$. The difference in runtime between our per-block generation and [GKS17] increases with n, but it can be observed that it is only a constant factor slower. The runtimes were always around an average factor of about 3.0 larger than that of [GKS17] (about 3.5 for $n = 21\,844$ and 2.4 for $n = 1\,398\,100$) with a deviation of at most 0.47 between the factors. For $n = 1\,398\,100$ [GKS17] runs in 1300 seconds as opposed to 3160 seconds for our per-block generation.

In Chapter 5, we describe our approach which uses disk space for storing information that has been stored earlier in memory: this implies a lot of file opening and closing, and writing and reading which takes more time and that is the reason for the constant factor slowdown.

6.2.4 Maximum Disk Space Usage

Figure 6.3 shows a histogram of maximum disk space used by the generated UC. In addition, it shows the extra disk space required for our per-block generation. In Section 5.1.3 we discuss our scalable way of generating files for UC generation which also involves deletion of unused files after every block generation. The disk space used by our generated UC files (outer skeleton and subgraphs) is the same as that for [GKS17] and therefore we denote this as UC disk space (green color). We mention that for every subgraph there is a maximum of 3 so-called tag files that store necessary information for generating subgraph nodes in the next

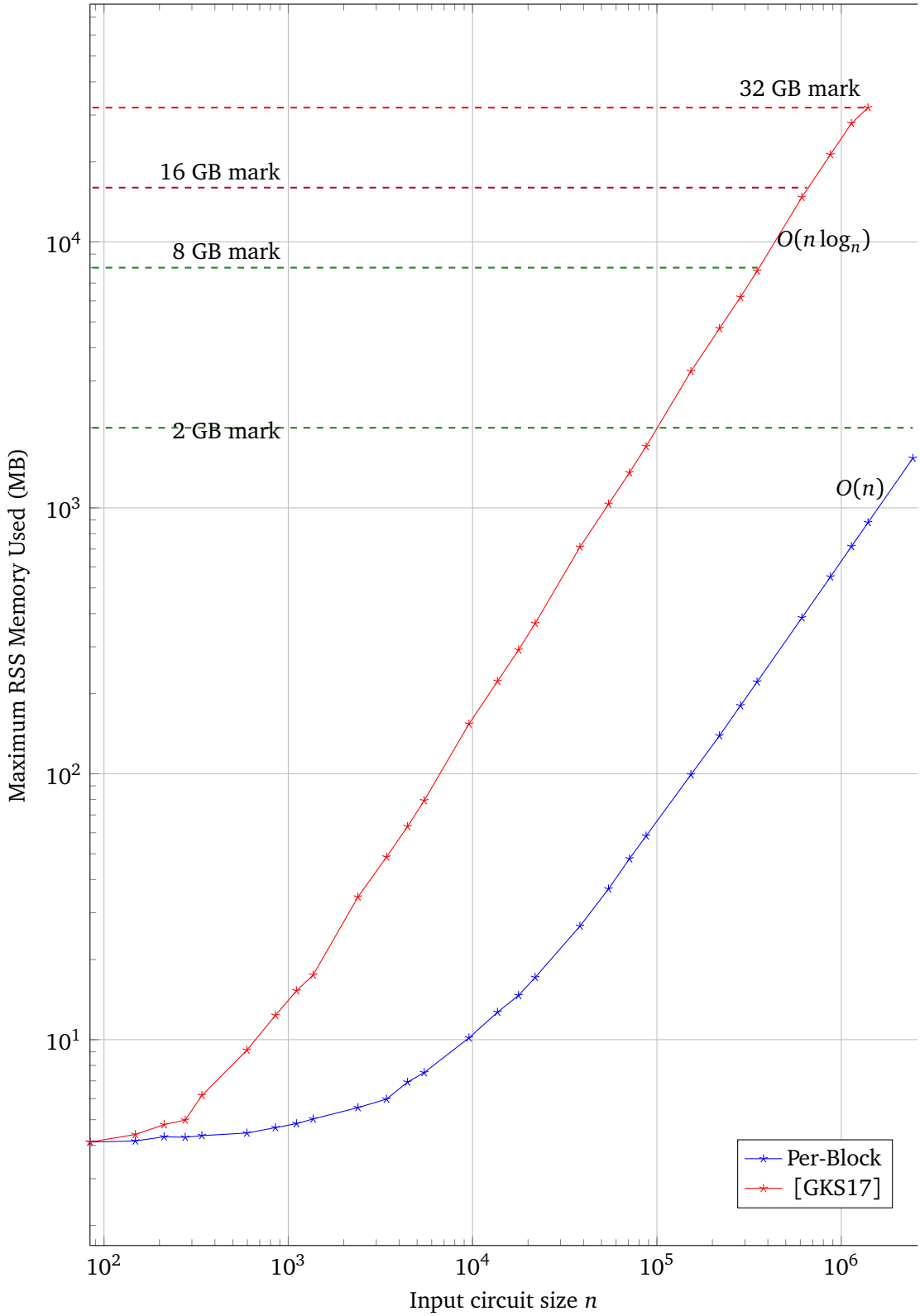


Figure 6.1: Comparison of the maximum Resident Set Size (RSS) memory used between our per-block and [GKS17]'s UC generation. [GKS17]'s implementation runs out of 32 GB of memory for $n > 1\,398\,100$ nodes.

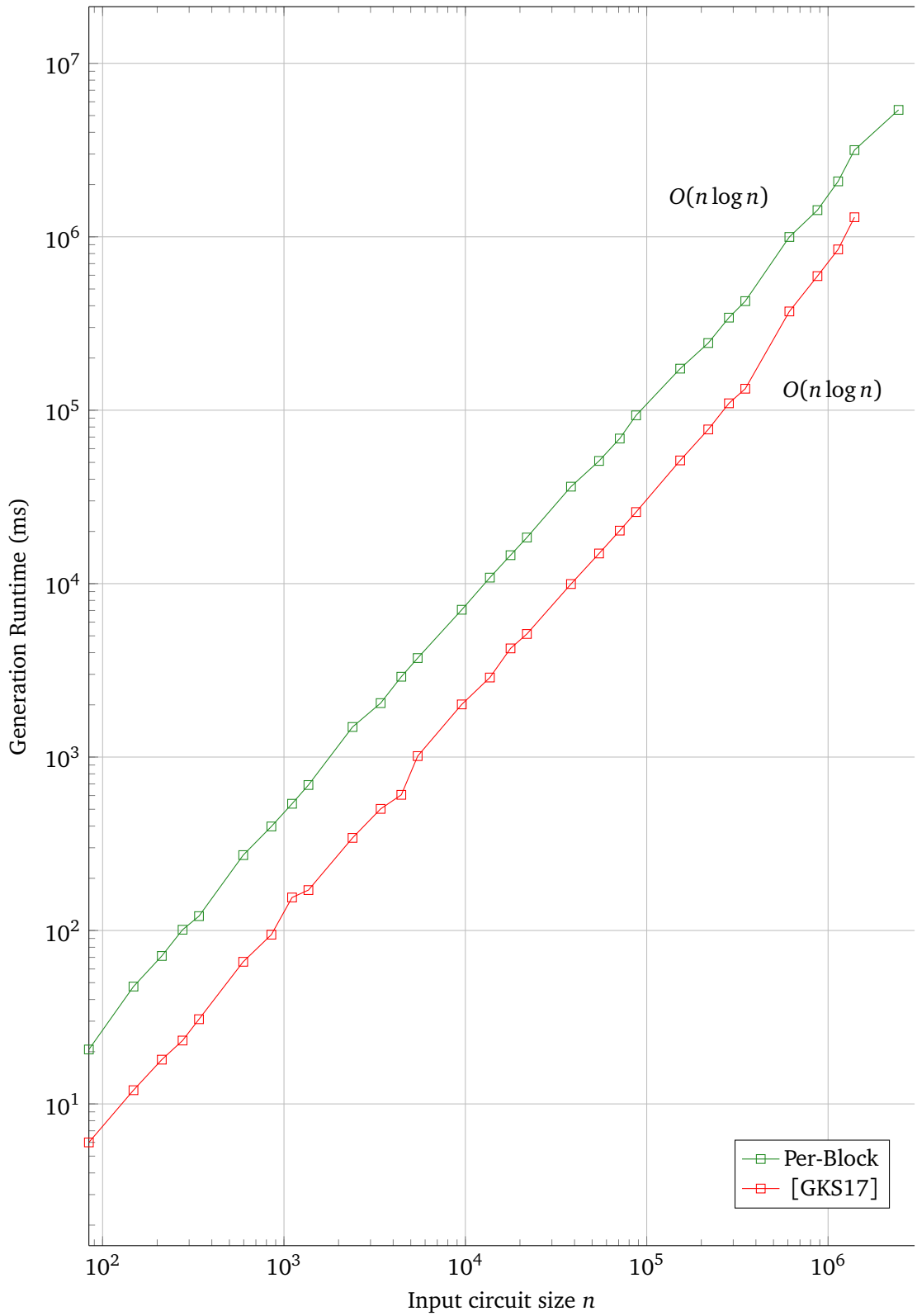


Figure 6.2: Comparison of the runtime of our per-block and [GKS17]'s UC generations for up to about $n = 2\,446\,000$ nodes.

2 per-block generations. The disk space used by the last three tag files is shown in the graph as extra disk space (pink color). The reason for this is that the largest disk space is used by the UC files, on top of which we only store the last tag files on disk before deleting them in the last step.

From the graph, the disk space occupied by our generation increases steadily and grows by a logarithmic factor larger ($O(n \log n)$) with the size of the UC. It is evident that the extra disk space required increases slowly and grows linearly in n. For a given $n = 2\,446\,676$, we require 7.3 GB for the UC files and only 145 MB of extra disk space for the tag files which is a relatively small storage space. The tag files can be larger in intermediate steps of the algorithm but never exceed the maximum disk space necessary when the UC files are also created. The UC files are the output of the algorithm and are designed to be as compact as possible by [GKS17]. Therefore, we indicate an at most 145 MB (for $n = 2\,446\,676$) of extra necessary disk space on top of the UC files for our implementation.

6.2.5 UC Size

For a k-way split UC construction with body (modular) block B_k, the asymptotic UC size can be calculated from $\text{size}(U_n^{(k)}(\Gamma_2)) = 2 \cdot \text{size}(U_n^{(k)}(\Gamma_1)) \approx 2 \cdot \frac{\text{size}(B_k)}{k} n \log_k n = 2 \cdot \frac{\text{size}(B_k)}{k \log_2(k)} n \log_2 n$ [LMS16]. [GKS17] recapitulate the size as $4.75 \cdot n \log n$ for 4-way split UCs.

For the calculation of the concrete size of the UC, we use the recursive formula from [GKS17] which calculates the size of the UC using only the AND gates which is rather of more interest in private function evaluation because of free XOR gates optimization [KS08b]. We document the sizes of the generated UCs in Table 6.1. The UC size grows by a logarithmic factor larger ($O(n \log n)$) with n [KS16].

Table 6.1 gives a summary of the results of our experiments. For some selected n numbers (big numbers from our experiments), we compare the memory consumption and the UC generation runtime between [GKS17] and our per-block generation. We also state the UC size and the total disk space used. We emphasize the more efficient version of a given benchmark between the two with bold text.

6.2.6 Summary

In summary, our per-block UC generation is far superior when it comes to memory consumption than the generation of [GKS17] with only about 1 537 MB of memory used for $n = 2\,446\,676$, whereas [GKS17]'s heavy dependence on memory resulted in a bad_alloc fault from using up all of the allocated memory. This improvement makes it more suitable for computation on devices with limited amount of memory, for instance mobile devices. Also, it can be used in high performance computing of Boolean circuits with millions of nodes. Our per-block UC generation, is, however, slower than the generation of [GKS17] by a constant factor of about 3.0 (cf. Section 6.1.3), which is acceptable. UC generation, however, is a

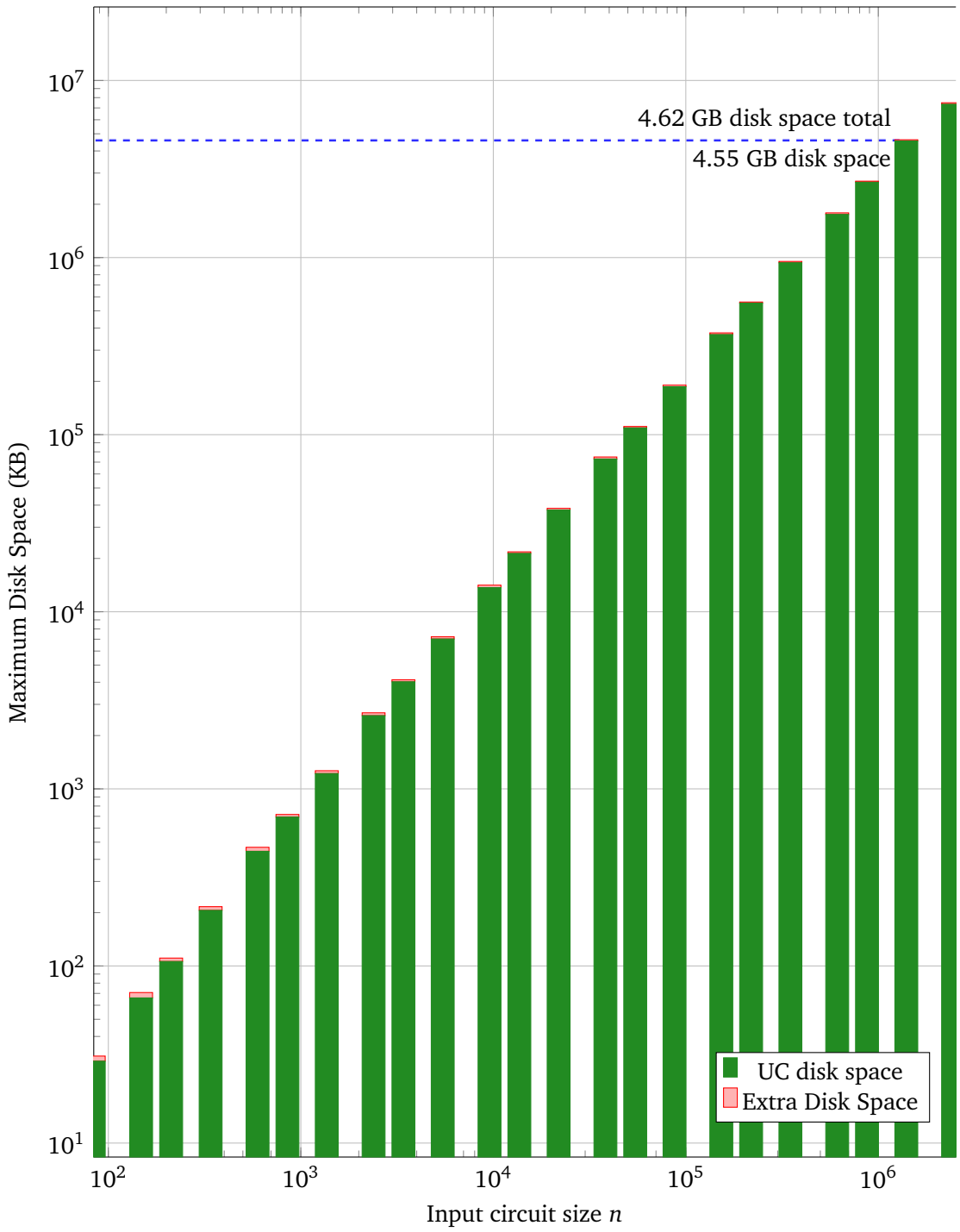


Figure 6.3: Disk space usage of our per-block implementation and extra disk space required (for the last 3 tag files) with up to almost 1 400 000 poles. The disk space used for the generated UC (excluding the tag files) for both our per-block and [GKS17]'s generation are the same.

n	Size (# AND gates)	Memory Used (MB)		UC Generation (ms)		Total Disk Space (MB)
		[GKS17]	Per-Block	[GKS17]	Per-Block	
13 652	$6.8 \cdot 10^5$	222.7	**12.7**	**2 875**	$10.8 \cdot 10^3$	21 798
152 916	$10.9 \cdot 10^6$	3258.7	**99.2**	**51 334**	$17.4 \cdot 10^4$	375 816
349 524	$26.3 \cdot 10^6$	7763.8	**221.3**	**133 023**	$42.6 \cdot 10^4$	951 282
611 668	$49.4 \cdot 10^6$	14717.1	**385.9**	**371 076**	$10 \cdot 10^5$	1 788 402
1 135 956	$94.7 \cdot 10^6$	27905.2	**715.7**	**845 750**	$2.1 \cdot 10^6$	3 646 839
1 398 100	$118.4 \cdot 10^6$	31989.9	**880.2**	**1 296 210**	$3.2 \cdot 10^6$	4 626 046
2 446 676	$220.7 \cdot 10^6$	*bad_alloc* error	**1 536.8**	*bad_alloc* error	$5.4 \cdot 10^6$	7 484 849

Table 6.1: Comparison of the memory consumption and generation of the UCs (per-block and [GKS17]) for some selected circuit sizes. Bold numbers denote which of the two implementations is efficient. The UC size is the same for both implementations.

precomputation step in most applications and therefore, a constant factor of runtime increase is not going to become a bottleneck. We iterate that multiple instances of our per-block UC generation can be run in parallel on big machines and even for memory-restricted devices, a UC for a circuit with about 1 million nodes can be generated.

7 Conclusion and Future Work

In this chapter we conclude our work and propose some future improvements in Sections 7.1 and 7.2 respectively.

7.1 Conclusion

Recent implementations of Valiant's 2-way and 4-way split UC constructions have an upper bound due to much dependency on memory [KS16; GKS17]. For the realization of real-world Private Function Evaluation applications, such as home automation or smart devices and remote diagnosis of smart cars, UCs have to be constructed for circuits with potentially millions of nodes. We introduce a per-block UC generation that makes this possible. Our scalable file generation makes it particularly intuitive to do edge-embedding of the generated UC. Günther et al.'s modular 4-way split UC generation imposes an upper bound on the size of the UC that can be generated due to memory at about 1 398 100. If the edge-embedding is done in addition, the upper bound due to the additionally stored supergraph as described in [GKS17]. With our per-block UC generation, we can generate UCs for Boolean circuits with millions of nodes.

Our generation is very efficient and uses very little memory as shown in Section 6.2. In addition, the extra disk space used for the generated UC is not much compared to [GKS17] as shown in Section 6.2. However, our per-block UC generation is slow compared to Günther et al.'s by an average runtime factor of 3.0 but in applications of PFE (which we are most concerned about), the UC generation is done offline (precomputed) and therefore we only account for offline computation costs.

PFE in large-scale real-world applications like home automation, remote diagnostics of smart cars, automobile insurance checking and remote patient monitoring are now a possibility with our per-block UC generation. Clients of applications can now be certain that their sensitive data cannot be read by the owners of the applications.

7.2 Future Work

In this thesis, we have explored the scalability of Valiant's Universal Circuits construction using a per-block approach which relies on very little use of memory and the file system with the pur-

pose of achieving scalability in order to generate very large UCs. Our scalable UC generation is particularly useful for generating large UCs for scalable private function evaluation. Smart car applications and diagnosis of automobiles for insurance protection and remote patient monitoring are a few motivational examples for large-scale PFE.

Besides improving the scalability of the UC generation, we designed our solution in such a manner that makes it extremely intuitive to perform the next step, the programming of the Universal Circuit, that is, edge-embedding into the universal graph. This can be done by recursively edge-embedding using the generated subgraph files and the outer skeleton (block edge-embedding) file for each corresponding graph of the Γ_2 graph (supergraph construction [KS16]). We iterate that this is made easy by our scalable file generation where each subgraph of the universal graph has its own file and can be easily identified (by its position and name) and associated to a corresponding Γ_1 graph from the Γ_2 graph (supergraph) as discussed in Section 4.3.2 and depicted in Listings 4.5 and 4.6. As a future work, the edge-embedding of the graph can be implemented. We give an idea of how this can be done in Section 4.3.

An algorithmic optimization would be to write all node descriptions of the outer skeleton and subgraph nodes (without actually creating the nodes) into their respective outer skeleton and subgraph files for each subgraph layer (cf. Section 5.1.2) without also assigning the topological numbers and additional wires. This would improve the runtime of the UC generation to numbers close to [GKS17] or even better but would require a lot of additional effort from the training of a sophisticated Deep Neural Network (DNN) (machine learning model).

A given UC generation is deterministic and therefore one can leverage data parallelism (in cases of high performance computing) where each cluster is responsible for generating a set of subgraph files. This approach can be coupled with the neural network approach and could lead to very good runtime improvements. This might lead to a slight increase in the usage of resources so there could be a limit set on the use of resources such as memory by clusters.

List of Figures

List of Tables

List of Abbreviations

UC Universal Circuit

UCs Universal Circuits

UAC Universal Arithmetic Circuits

SFDL Secure Function Definition Language

SHDL Secure Hardware Definition Language

EUG Edge-Universal Graph

DAG Directed Acyclic Graph

DAGs Directed Acyclic Graphs

SFE Secure Function Evaluation

OT Oblivious Transfer

GMW Goldreich-Micali-Wigderson

PFE Private Function Evaluation

BMR Beaver-Micali-Rogaway

DFS Depth-First Search

RSS Resident Set Size

RAM Random Access Memory

Bibliography

[ALSZ13] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, M. ZOHNER. **"More efficient oblivious transfer and extensions for faster secure computation"**. In: *CCS*. ACM, 2013, pp. 535–548 (cit. on pp. 8 sq.).

[ALSZ15] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, M. ZOHNER. **"More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries."** In: *EUROCRYPT*. LNCS 9056 (2015), pp. 673–701 (cit. on p. 8).

[Bea96] D. BEAVER. **"Correlated Pseudorandomness and the Complexity of Private Computations"**. In: *STOC*. ACM, 1996, pp. 479–488 (cit. on p. 8).

[BFK+09] M. BARNI, P. FAILLA, V. KOLESNIKOV, R. LAZZERETTI, A. SADEGHI, T. SCHNEIDER. **"Secure Evaluation of Private Linear Branching Programs with Medical Applications"**. In: *ESORICS*. Vol. 5789. LNCS. Springer, 2009, pp. 424–439 (cit. on p. 1).

[BMR90] D. BEAVER, S. MICALI, P. ROGAWAY. **"The Round Complexity of Secure Protocols (Extended Abstract)"**. In: *STOC*. ACM, 1990, pp. 503–513 (cit. on p. 9).

[BNP08] A. BEN-DAVID, N. NISAN, B. PINKAS. **"FairplayMP: a system for secure multi-party computation"**. In: *CCS*. ACM, 2008, pp. 257–266 (cit. on p. 7).

[BPSW07] J. BRICKELL, D. E. PORTER, V. SHMATIKOV, E. WITCHEL. **"Privacy-preserving remote diagnostics"**. In: *CCS*. ACM, 2007, pp. 498–507 (cit. on p. 1).

[GKS17] D. GÜNTHER, Á. KISS, T. SCHNEIDER. **"More Efficient Universal Circuit Constructions"**. In: *ASIACRYPT*. Vol. 10625. LNCS. Springer, 2017, pp. 443–470 (cit. on pp. 1 sqq., 13 sqq., 18 sq., 21, 23, 25, 27, 30–34, 39 sqq., 45, 47, 52 sq., 55–65).

[GMW87] O. GOLDREICH, S. MICALI, A. WIGDERSON. **"How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority"**. In: *STOC*. ACM, 1987, pp. 218–229 (cit. on pp. 7, 9).

[IKNP03] Y. ISHAI, J. KILIAN, K. NISSIM, E. PETRANK. **"Extending Oblivious Transfers Efficiently"**. In: *CRYPTO*. Vol. 2729. LNCS. Springer, 2003, pp. 145–161 (cit. on p. 8).

[KR11] S. KAMARA, M. RAYKOVA. **"Secure outsourced computation in a multi-tenant cloud"**. In: *IBM Workshop on Cryptography and Security in Clouds*. 2011, pp. 15–16 (cit. on p. 1).

[KS08a] V. KOLESNIKOV, T. SCHNEIDER. **"A Practical Universal Circuit Construction and Secure Evaluation of Private Functions"**. In: *FC*. Vol. 5143. LNCS. Springer, 2008, pp. 83–97 (cit. on pp. 3, 14, 20 sq., 23).

[KS08b] V. KOLESNIKOV, T. SCHNEIDER. **"Improved Garbled Circuit: Free XOR Gates and Applications"**. In: *ICALP*. Vol. 5126. LNCS. Springer, 2008, pp. 486–498 (cit. on pp. 9, 61).

[KS16] Á. KISS, T. SCHNEIDER. **"Valiant's Universal Circuit is Practical"**. In: *EUROCRYPT*. Vol. 9665. LNCS. Springer, 2016, pp. 699–728 (cit. on pp. 1 sqq., 5 sq., 14 sq., 17 sqq., 21, 23, 31, 39, 45 sq., 61, 64 sq.).

[LMS16] H. LIPMAA, P. MOHASSEL, S. S. SADEGHIAN. **"Valiant's Universal Circuit: Improvements, Implementation, and Applications."** In: *IACR Cryptology ePrint Archive* 2016/17 (2016) (cit. on pp. 1, 3, 14 sq., 18, 20 sq., 23, 40, 61).

[LP09] L. LOVÁSZ, M. D. PLUMMER. *Matching theory*. Vol. 367. American Mathematical Society, 2009 (cit. on p. 18).

[MNPS04] D. MALKHI, N. NISAN, B. PINKAS, Y. SELLA. **"Fairplay - Secure Two-Party Computation System"**. In: *USENIX Security Symposium*. USENIX, 2004, pp. 287–302 (cit. on p. 7).

[MS13] P. MOHASSEL, S. S. SADEGHIAN. **"How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation"**. In: *EUROCRYPT*. Vol. 7881. LNCS. Springer, 2013, pp. 557–574 (cit. on p. 13).

[MSS14] P. MOHASSEL, S. S. SADEGHIAN, N. P. SMART. **"Actively Secure Private Function Evaluation"**. In: *ASIACRYPT*. Vol. 8874. LNCS. Springer, 2014, pp. 486–505 (cit. on p. 13).

[OI05] R. OSTROVSKY, W. E. S. III. **"Private Searching on Streaming Data"**. In: *CRYPTO*. Vol. 3621. Lecture Notes in Computer Science. Springer, 2005, pp. 223–240 (cit. on p. 1).

[RB12] M. D. RIEDEL, J. BRUCK. **"Cyclic Boolean circuits"**. In: *Discrete Applied Mathematics* 160.13-14 (2012), pp. 1877–1900 (cit. on p. 6).

[Reu06] L. REUTHER. "Disk storage and file systems with quality of service guarantees". PhD thesis. Dresden University of Technology, Germany, 2006 (cit. on p. 11).

[Sch08] T. SCHNEIDER. **"Practical Secure Function Evaluation"**. In: *Informatiktage*. Vol. S-6. LNI. GI, 2008, pp. 37–40 (cit. on pp. 6, 14).

[Sha+49] C. SHANNON. **"The synthesis of two-terminal switching circuits"**. In: *Bell Labs Technical Journal* 28.1 (1949), pp. 59–98 (cit. on p. 6).

[SZ13] T. SCHNEIDER, M. ZOHNER. **"GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits"**. In: *FC*. Vol. 7859. LNCS. Springer, 2013, pp. 275–292 (cit. on pp. 9 sq.).

[Val76] L. G. VALIANT. **"Universal Circuits (Preliminary Report)"**. In: *STOC*. ACM, 1976, pp. 196–203 (cit. on pp. 1, 3, 6, 13 sqq., 17 sqq., 21, 25, 39, 56).

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

[Yao86] A. C.-C. YAO. **"How to Generate and Exchange Secrets"**. In: *FOCS*. IEEE, 1986, pp. 162–167 (cit. on pp. 7 sq.).