TECHNISCHE
UNIVERSITÄT
DARMSTADT

Bachelor Thesis

Private Function Evaluation for Multi In- and Output-Gate Circuits

Maximilian Stillger

June 8, 2022

ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

Cryptography and Privacy Engineering Group
Department of Computer Science
Technische Universität Darmstadt

Supervisors: M.Sc. Daniel Günther
Prof. Dr.-Ing. Thomas Schneider

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Maximilian Stillger, die vorliegende Bachelor Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

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

I herewith formally declare that I, Maximilian Stillger, have written the submitted Bachelor 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.

Darmstadt, June 8, 2022

Maximilian Stillger

Abstract

Private Function Evaluation (PFE) is a cryptographic protocol, which allows two or more parties to securely compute a function, such that only one party knows the function and every party can contribute inputs to the function without revealing them. The only revealed information is the function output. Less secure approaches, which are referred to as *Semi Private Function Evaluation* (SPFE), may leak some metainformation of the function, by which the private function can be narrowed down to a smaller subset of all functions, while improving the performance of the protocol (memory consumption, time of evaluation etc). Several different approaches for PFE have already been proposed including a reduction to Secure Function Evaluation (SFE), which evaluates a public function on private inputs.

This is achieved by constructing special public functions called *Universal Circuits* (UC) which can be programmed to represent an arbitrary boolean circuit up to a certain size, thus hiding the private function inside of the UC as programming information. Past constructions of UCs used circuits with gates of at most 2 inputs and 1 output (Boolean AND, OR, XOR gates etc). With the advent of more efficient evaluation of larger Look Up Tables (LUTs) we may improve the size and evaluation cost of our Universal Circuits by replacing the Universal Gates of [AGKS20] by larger *Universal LUTs* (ULUT), programmable LUTs with multiple inputs and outputs. Thus we propose two modified UC constructions, based upon the constructions of [Wig21], which allowed for the embedding of circuits with multi input-single output LUTs. The *fixed construction*, which accomodates multi input-multi output ULUTs with a parameter k for the maximum number of inputs and outputs per ULUT, is fully private up to the amount of LUTs used and the parameter k. The *dynamic construction*, which leaks more detailed arity information of each ULUT, yields a smaller UC size and can thus be evaluated cheaper in MPC. We argue the correctness of these constructions and give complexity bounds of their size. We achieve an overall size improvement relative to [Wig21] for most of the tested circuits in the fixed construction, and in all tested circuits a a significant ($\approx 22\%$) improvement with the dynamic construction.

Acknowledgments

I would especially like to thank Daniel Günther, Hossein Yamale, Arthur Wigandt, Joachim Schmidt and Prof. Dr. Thomas Schneider for teaching and supporting me in the matter of private function evaluation and specific protocols and constructions to improve performance, supporting me in matters of technical complications and giving feedback that was critical for my work.

Contents

1 Introduction

Private Function Evaluation (PFE) is a cryptographic protocol, which allows two or more parties to compute a function collaboratively so that only one of the parties knows the exact function to be computed and all parties only know their own input to the function. The result of the function can be revealed to an arbitrary subset of the involved parties. Without loss of generality, we restrict the class of functions we want to be able to compute in PFE to combinatorial boolean circuits. PFE can be weakened to Semi Private Function Evaluation (SPFE) [PSS09; GKSS19], where we leak further aspects of the function, like more specific size information or entire parts of a function. This may be useful as it requires less resources to compute parts of the function that do not reveal important segments of it.

Private Function Evaluation can be accomplished by creating a special boolean circuit called a *Universal Circuit* (UC) [Val76]. A Universal Circuit can embed an arbitrary combinatorial Boolean function f up to some some size $n = u + g + v$ where u is the number of inputs, g the number of gates and v the number of outputs. The function f is then represented in terms of a string p_f called the *programming* of the universal circuit, which when given to the programming inputs of the UC causes the UC to behave exactly as the function f on input x:

$$UC(p_f, x) = f(x).$$

If the universal circuit is now computed with a secure multi-party computation (SMPC) protocol such as Yao's Garbled Circuit [Yao86] or the GMW protocol [GMW87], the programming information is hidden as a secret input along with the function inputs, thus fullfilling the privacy of the function by hiding its circuit topology and exact gates.

Some classes of functionalities, e.g., ripple carry adders or the AES block cipher make great use of modularization (a ripple carry adder can be built from full adders and AES from substitution boxes) to accomplish optimized circuit sizes and easily definable wire interfaces between the submodules. Note also that any boolean circuit may be represented by Look Up Tables instead of primitive gates like AND, OR, etc.

Previously in the work of [Wig21], PFE via UCs with multi input-single output LUTs has been implemented, such that either all universal gates in a circuit have the same number of inputs, or that the number of inputs dynamically adapts, in which case we only have SPFE, as we can infer the arity of each LUT of the circuit, which may allow us to infer some information about the circuit topology.

In this work, we want to exploit the benefits of abstraction into submodules with multiple inputs *and multiple outputs* by providing a notion of encapsulation of larger substructures into singular gates inside of universal circuits. These gates which we refer to as *Universal Look Up Tables* (ULUTs) accommodate multiple inputs as well as multiple outputs and are programmable in their look up tables. This improvement requires modifications of the two previous construction algorithms of [Wig21], which only embedded functions where each gate had multiple inputs but at most 1 output (which could be used multiple times), in order to support multiple different outputs per LUT. The first construction is called the fixed construction achieving full privacy of the function while only leaking the size of the circuit and maximum input and output arity of the gates occurring in it. The second construction, called dynamic, reveals more precise arity information about each gate but still hides the topology of the entire function, which leaks less information than previous SPFE frameworks and yields a size improvement. Reducing the size of a universal circuit is the primary target of our optimization, as the amount of AND gates inside of the universal circuit dictates the amount of memory and communication overhead required in the MPC protocol (in Yao's protocol [Yao86], each AND gate requires a garbled table and thus more information to be transmitted).

1.1 Related Work

Secure Multi Party Computation (MPC) was first proposed by [Yao86] to solve the millionaires' problem. The first recursive construction of Universal Circuits (UC), based on so called Edge Universal Graphs (see §2.0.5), was given by [Val76]. The construction of these Edge Universal Graph-based UCs was improved significantly by [LYZ+21], reducing the asymptotic complexity from $\approx 4.5n \log n$ to $\approx 3n \log n$ where n is the size of the embedded circuit. [KS16; AGKS20; GKS17; ZYZL19] modularized the construction, reducing the memory overhead of the UC compiler and allowing for further combination of different construction methods into hybrid UCs with adaptable recursion parameters.

[KS08; SS09] already proposed different constructions for more specialized UCs with multiple (d) inputs and one output, which obliviously route bundles of input wires between the gates. Their first iterative construction has complexity $\mathcal{O}(n^2 \log d)$. Their second construction uses three types of modules (input selection, output selection and a generalized universal blocks) and achieves complexity $\mathcal{O}(dn \log^2 n)$. The third construction uses the recursive nature of [Val76] and modifies a circuit to have a fanin and fanout of at most d with some overhead. The first two constructions are more beneficial for smaller circuits to be embedded, while the third construction may be used for larger circuits, and is related to our work in the sense that in it a circuits fanout is reduced before passing it to a d-way generalization of a classical construction like Valiant's. [Wig21] provides a UC compiler upon which our improvement is built. It already addresses the creation of multiple inputs for the use of Look Up Tables with a single output while including an implementation of [LYZ+21]. It provides a fixed construction, which parallelizes classical constructions on the EUG level, aswell as a dynamic construction, which uses pre-processing of the circuit to be embedded and post-processing of the universal circuit to better adapt to circuits with LUTs of varying input sizes.

Our work extends both the fixed and dynamic construction of [Wig21] to allow for an arbitrary amount of distinct outputs for each LUT and also adapts the post-processing for the dynamic approach for varying output sizes.

More generally, other approaches for PFE have already been proposed:

1. Based on Partial Homomorphic Encryption [KM11], this yields a linear complexity in the size of the circuit (instead of $n \log n$), while still requiring homomorphic encryption, which is more expensive to instantiate than symmetric encryption. This approach was optimized and implemented in a highly efficient manner by [HKRS20].
2. Permutation Networks [MS13; MSS14], which use oblivious permutations to hide circuit topologies in a computation and separately securely evaluate gates on the permuted inputs in rounds. They operate with asymptotic complexity $n \log n$ and only require symmetric encryption.
3. Trusted Execution Environments, which are specific subcomponents of computer processors (e.g. Intel SGX Platform) may be used as a secure enclave inside of one of the parties for computing special cryptographic functionalities. [FKSW19] demonstrated efficient PFE which was resilient against side channel attacks, by evaluating UCs inside of a SFE protocol that was adapted specifically for the specialized hardware secured environment.

1.2 Applications

Private Function Evaluation can be of practical use in online computations that have algorithms, which are trade secrets (proprietary algorithms), at especial risk of security under leakage (information about the operational structure of institutions, which is a necessary part of the algorithm, but may allow for attacks on that institution) or inputs of the computation, that are protected under privacy law (sensitive personal information, address data) or that the party might not want to share (properties which do not uniquely identify that individual, but are still highly personal). For example institutional Credit Checking and Scoring of private individuals is a common and useful application [FAZ05] of PFE, as it hides trademarked algorithms determining insurance/credit rates as well as sensitive transaction histories and personal information of customers. In [SS09] the more specific usage of UCs with multiple inputs was shown for the evaluation of Neural Networks, while hiding the exact topology of the neural network. Universal Circuits may be used in other cryptographic applications like improving the Verifier workload in non interactive zero knowledge proofs [GGPR13] or performing attribute-based encryption with circuits [GVW15]. [FVK+15] also demonstrates the usage of PFE for Database Management Systems which allow for querying of data and revealing neither the querying algorithm nor the entire dataset on which the query operates.

1.3 Contributions

We propose, based on the UC compiler of [Wig21], an improved fixed construction based on Look Up Tables with multiple (>2) inputs and multiple (>1) outputs (cf. § 4.3), thus further reducing size requirements and also improving computation based on specific LUT related SFE improvements [DKS+17]. This comes at the cost of revealing some information on the embedded function, more specifically the input and output arity information of LUTs used inside the circuit. We also propose an improved dynamic construction that leaks the fanin and fanout of each gate embedded (cf. § 4.4) and produces a UC that has smaller size but is only semi-private. Note that the two constructions rely on atleast one of the two EUG constructions presented in § 3.

We benchmark exemplary circuits for arithmetic (ripple carry adder, karatsuba multiplier cf. § 5.3) as well as some cryptographic example circuits (AES,DES, SHA 256 cf. §5.4) and compare the results with prior work. We outline the complexities of all the presented constructions and explain their correctness. We achieve an $\approx$ 35% size improvement over [Wig21] for circuits of arithmetic functions in the dynamic construction and a 18% improvement in the fixed construction. We achieve an improvement of $\approx$ 15% in cryptographic circuits tested. We outline the overall behavior of our construction methods under more general settings (cf. § 5.5). We also give a method for deriving size and fanout optimized multi input-multi output LUT based circuits from arbitrary combinatorial Boolean circuits (cf. § 5.1.2).

2 Preliminaries

In this section we introduce some mathematical tools to describe graphs before we can study the different UC constructions. The mathematical definitions are largely identical to those of [Wig21], but with slight simplifications for brevity's sake.

We denote all graphs by $G = (V, E)$ such that $E \subseteq V \times V$, where V denotes the set of vertices (or nodes) and E the edge relation on these vertices. Later we may also use the Notation $G = (V, E) = ((P, N), E)$, where $V = P \cup N$ with $N \cap P = \emptyset$. P describes the set of so called *poles* which are special nodes that represent the nodes to be embedded inside the graph of the UC and are connected by the remaining nodes N in paths, in order to hide their connections. We state for $G = (V, E)$ and $G' = (V', E')$ the subset relation as $G \subseteq G' \iff V \subseteq V' \wedge E \subseteq E'$. Distinguishing properties of graphs are both their arities, as well as occurring cycles, which allow us to make certain assertions about them, such as the embedding properties we will need in our constructions.

Definition 2.0.1. *We define the following arities of graphs.*

1. *the indegree of a vertex v is $\delta^{+}(v) = |\{u \in V : (u, v) \in E\}|$.*
2. *the outdegree of a vertex v is $\delta^{-}(v) = |\{u \in V : (v, u) \in E\}|$.*
3. *the in/outdegree of a graph is given by $\delta^{+/-}(G) = max\{\delta^{+,-}(v) | v \in G\}$.*

We also define the predicate *isPath* as follows.

Definition 2.0.2. *We define in a Graph $G = (V, E)$ for arbitrary $E' \subseteq E$ and $u, v \in G$:*

$$\mathit{isPath}(u, v, E') \iff \exists n \in \mathbb{N} : \exists_{i=1}^{n} e_i = (u_i, v_i) \in E' : u_1 = u \wedge v_n = v \wedge \forall j < n : v_i = u_{i+1}$$

isPath states, that there is a chain of edges in the defined subset, such that the first edge starts at vertex u and the last edge ends at vertex v.

Definition 2.0.3. *A directed acyclic graph (DAG) is a graph $G = (V, E)$ in which the edge relation is strictly irreflexive and antisymmetric*

$$\forall x, y \in V : ((x, y) \in E \rightarrow \neg((y, x) \in E)) \wedge \forall x \in V : \neg((x, x) \in E).$$

A DAG may not include paths (sequences of edges) that contain a vertex twice, i.e, it has not cycles.

$$\forall u \in V : \forall E' \subseteq E : \neg(isPath(u,u,E')).$$

We denote the class of all DAGs with $\max(\delta^-(G), \delta^+(G)) = \rho$ *and* $|V| = n$ *as* $\Gamma_\rho(n)$.

Another important property of graphs is their so called *edge coloring index*. We use it in the UC constructions to prove, that the task of edge embedding a graph can be divided k sub problems, such that each sub problem only needs to edge embed a sub graph with special fanin/fanout and combining the sub solutions in a non interfering way. The parameter k is then precisely this edge coloring index.

Definition 2.0.4. *Let* $G = (V,E)$ *be an arbitrary graph. We define the edge coloring index as:*

$$\chi(G) = \min\{|image(f)| | f \in Colorings(G)\}.$$

where we define the colorings by:

$$Colorings(G) = \{f : V \times V \to \mathbb{N} | \forall x,y,z \in V : ((x,y) \in E \wedge (x,z) \in E) \to (f(x,y) \neq f(x,z))\}$$

i.e., $\chi(G)$ *is the smallest amount of colors necessary to color all adjacent edges differently.*

A first important basic result for DAGs is that we can bound their edge coloring index (the amount of colors necessary to color all adjacent edges pairwise differently) by the fanin/fanout as follows. The proof and derived coloring algorithm was given in [Die17], with the proof slightly modified to be more informal.

Theorem 2.0.1. *The following equivalent statements hold:*

1. *König-Hall Theorem [Die17]: Any* $G' \in \Gamma_\rho(n)$ *has at least one* ρ *coloring.*
2. *Any bipartite* G *has* $\chi(G) \leq \Delta(G)$ *where* $\Delta(G) = \max(\delta^-(v) + \delta^+(v))$.

Note that the two statements are equivalent, because edges that are adjacent in G' will be adjacent in an associated product graph that is bipartite. We will describe this graph in Theorem 2.0.2. Since the neighboring can be preserved between these two graphs, the coloring index is also preserved.

Proof. The case in which more colors than the maximum degree are available is trivially solvable as we can enumerate all edges of some node and pick a new color for each adjacent edge without a conflict in colors. The opposite case where we constraint the number of colors now to be at most the maximum degree is less trivial. We prove by induction over the cardinality of the edge set.

Case 1 (Induction Base)**.** *Assume any bipartite* G *with* $|E| = 1$.

Here we simply color the one edge with the one color we must use.Therefore $\Delta = 1$ and $\chi(G) = 1$ which suffices.

Case 2 (Induction Step)**.** *As Induction Hypothesis we assume, that we can color any bipartite G with $|E| = m$ with $\chi(G) \leq \Delta$.*

We now want to color a bipartite supergraph $G' = (V', E'), G \subseteq G'$ such that $|E'| = m + 1$.

We remove an arbitrary edge $e' = (u, v)$ such that we get a smaller edge set. By induction hypothesis we now get a fitting coloring of this smaller graph with Δ colors, since we removed one edge and the graph is still bipartite.

We want to add the edge back that we removed previously, while adjusting the coloring. Note that $\delta^+(u), \delta^-(v) \leq \Delta - 1$ and thus each of the nodes has at least one unused color. By case distinction u and v both have either at least one shared unused color or each of them has at least one distinct unused color, while one of them is used by the other.

In the former case we assign that color to e' and thus get a full coloring, closing the Induction.

In the latter case we need to slightly change the prior coloring of the graph. Then there exists a color β so that for some edge $(u, *) \rightarrow \beta$ and for no $(*, v) \rightarrow \beta$ and a color $\alpha \neq \beta$ that has no $(u, *) \rightarrow \alpha$. Now there must exist at least one longest alternating $\alpha - \beta$ path starting with that β edge e^* $(e^* \rightarrow \beta)$. We also note that all β edges point from one side of the bipartite graph to the other while all α edges inside the walk vice versa, by the nature of bipartite graphs and the alternating path.

We thus may conclude that there are no cycles in this path, as they would either induce a wrongful coloring (which by induction hypothesis could not exist) as well as no loop with u as this would include an α edge starting in u which could not exist by our choice of α. Also v $(\neq u)$ may not be in this path because as it sits on the opposite side of u it must have a β edge incoming which was also not possible. Therefore this longest possible path with alternating edges starting with β is entirely filled with unique nodes and may not contain v only u. If we now flip all the colors on this path (α to β and vice versa) we still have a valid coloring of the graph as every node only uses α and β at most once and still does afterwards. If we color the back added edge β this is possible because we swapped e^* to α and v never had a β edge.

Note that while we provided the argument here where u was implicitly on the left side this may be arbitrarily switched and the argument stays the same. $\square$

This proof yields by proof mining the natural Algorithm 2.1 for edge coloring a DAG, where the first case of the induction step represents the greedy choice of new colors for the graph and the second case follows this up by a subsequent correction step.

On a side node, it is trivial to see why a bipartite graph with in- and outdegree 2 can be 2-colored by coloring an initial edge randomly, and coloring all subsequent edges with alternating colors via depth-first search, thus having an algorithm that can be performed very fast. This algorithm can be further improved by a plethora of (probabilistic) graph algorithms

Algorithm 2.1 $k-edge-color(G)$

```
      Input:   bipartite graph $G = (V, E)$
      Output:  $k$ edge coloring of $G$
 1: $\Delta \leftarrow \max_{v \in G}\{deg_G(v)\}$
 2: for Edge $e = (u, v) \in E$ without color do
 3:     if there is a color $i \le \Delta$ that does not occur in any of the neighboring edges of $e$ then
 4:         $e.color \leftarrow i$
 5:     else
 6:         Choose $\beta$ s.t. $e'.color = \beta$ for some $e' = (u, v')$ and $\forall e^* = (u', v) : e^*.color \ne \beta$
 7:         Choose $\alpha \ne \beta$ s.t. it is not used by edge going to $u$
 8:         Swap colors of longest $\beta - \alpha$ alternating path starting with $e'$
 9:         $e.color \leftarrow \beta$
10:     end if
11: end for
```

(e.g. [Alo03]), as the edge-coloring-problem in general is well understood. We now use the previously attained result in order to partition a DAG into several DAGs of lower in- and outdegree.

Lemma 2.0.2. *Every graph $G \in \Gamma_\rho(n)$ can be partitioned into ρ $\Gamma_1(n)$ sub graphs.*

Proof. Let $G = (V, E)$. First we build a special product graph of our original graph $G' = (V', E')$ such that

$$V' = V \cup \{v' : v \in V\}.$$

$$E' = \{(u, v')' : (u, v) \in E \wedge v' \text{ is the copy of v as above}\}.$$

This transformation amounts to duplicating the nodes and relaying the edges to the new right side of the bipartite graph. We now utilize the algorithm described in the previous Theorem 2.0.1. Since every node in the bipartite graph has only in- or outgoing edges and thus at most degree ρ we can color the edges of bipartite graph with at most ρ colors. Now to get the final partition we setup for $i \in [\rho]$.

$$G_i = (V_i, E_i), V_i = V E_i = \{e \in E : \chi(e') = i\}.$$

i.e., we create one copy of the original graph for every color, whilst only including the edges of color i in the i-th copy-graph. Since every edge has exactly one color and every color has one sub graph we have

$$\bigcup_{1 \le i \le \rho} G_i = G.$$

Note also that due to the coloring restriction, every node has exactly in/outdegree at most 1 (else we would have a coloring violation). □

These partition graphs are later used in all our constructions, as every partition graph of the DAG to be embedded into an EUG will get mapped into its own copy of an Valiant or Liu EUG, which by construction each may only accommodate graph with fanin 1.

We will now define EUGs.

Definition 2.0.5. *An* Edge Universal Graph *on the class $\Gamma_\rho(n)$ ($\rho -$**EUG**) is a special graph $G = ((P, N), E)$ such that for every DAG $G' = (P, E') \in \Gamma_\rho(n)$ it holds, that for any $e_1 = (p_i, p_j) \neq e_2 = (p_k, p_l) \in E'$ with $i < j \wedge k < l$ according to topological order:*

$$\exists E_1, E_2 \subseteq E' : (E_1 \cap E_2 = \emptyset) \wedge \mathit{isPath}(p_i, p_j, E_1) \wedge \mathit{isPath}(p_k, p_l, E_2).$$

i.e., every DAG with at most n nodes, can be embedded with edge disjoint paths into the EUG, based on the topological order of the graph. The set P of distinguished nodes defines the set of pole nodes, which represent the original nodes of the graph to be embedded. Alternatively, one may define the embedding property by maps from an arbitrary vertex set to the poles, but this is only a slight notational difference which would make the correctness proof more difficult (but is more natural from an implementation understanding). It should also be noted here that, while the DAG to be embedded has to be acyclic by definition, the same does not necessarily hold for the EUG. This is used by the construction of [LYZ+21] in order to further improve the size of EUGs. Finally an EUG with cycles has to be modified into an acyclic one too however, as it has to be instantiable as a boolean circuit in MPC protocols.

As it would be unwieldy to produce ρ-grade EUGs immediately, we instead construct just 1-EUGs and merge them. This works in tandem with the partitioning of the DAG outlined earlier.

Theorem 2.0.3. *A ρ-EUG $G = ((P, N), E)$ and a σ-EUG $G' = ((P, N'), E')$ can be merged into a $(\rho + \sigma)$-EUG $G'' = ((P, N''), E'')$.*

Proof. We simply merge the EUGs at the poles, i.e., $G'' = ((P, N \cup N'), E \cup E')$. If we now have a DAG $G* \in \Gamma_{\rho+\sigma}(n)$ then for any vertex we route the first ρ outgoing edges over the same edges as in G and the consecutive σ possible edges over the same edges as in G'. Since the two edge sets are disjoint, the new paths must also always be disjoint. We also know that all possible $\rho + \sigma$ edges could be embedded as they were well defined previously by the edge embeddings of the original EUGs, because both EUGs had the same set of poles that could be connected. $\square$

3 Related Work

In this section we give an overview of the necessary constructions for Edge Universal Graphs (EUG) to produce UCs. We consider the simpler Valiant 2-way construction [Val76] to understand the basic principle of embedding in an easier setting, as well as the more complex Liu et al's [LYZ+21] 2-way construction, which produces a smaller EUG size. The construction of Liu et al. is based on Valiants construction, and works very similar in both the structure of recursion and the embedding, but makes use of an adjustment to cylical graphs in order to achieve a smaller size. We also briefly discuss miscellaneous constructions and explain the classical conversion process from a 2-EUG with its embedding of a graph into a Universal Circuit with a programming of any circuit corresponding to the graph.

3.1 Valiants (2-Way) Construction [Val76]

Valiant's [Val76] 2-way Edge Universal Graph (EUG) construction with its subsequent UC construction is the original UC construction. Among UC constructions, the recursive construction approach is most beneficial if we consider circuits of large size ($> 10\,000$ gates) as its size complexity ($\mathcal{O}(n \log n)$) scales better than most other approaches. The construction however is outperformed e.g. by the iterative approach of [SS09] for small circuits (cf. § 3.3).

The basic concept behind Valiant's construction relies on the recursive solving of the embedding problem in smaller subgraphs, which may be considered as a representation of the *interface* of the nodes, we wish to embed. We recall that the graph representation of the input specific graph we want to embed into the EUG is built from n so called poles, which have only degree 1, as we have split the original graph into subgraphs with this fanin/fanout (cf. Theorem 2.0.2). We manually manufacture a small graph, called a k-way Superpole, which allows the universal connection from some set of k input nodes I to a set of poles P inside the Superpole, and subsequently connecting these poles to a set of k output nodes O such that any embedding from these input nodes to the poles and output nodes with in/outdegree 1 may be made. These Superpoles will essentially serve as the recursion anchor to our embedding problem, as we can memorize the correct embeddings for these small handcrafted graphs. The input and output nodes of the Superpole are also called recursion points because they are the interfaces to sub graphs of the EUG which serve the recursive embedding process.

Now we connect many Superpoles together into a so-called *skeleton* such that they form a chain connected at the recursion points. For n poles we will need $\lceil \frac{n}{k} \rceil$ Superpoles, as each Superpole will give us k poles to work with. Next, we repeat this skeleton construction recursively at the inner recursion points (we ignore the first and final recursion points). By this we use the recursion step of our last skeleton as the pole sets for the next subskeletons; we build the same chained Superpole construction but with the set of recursion points as the poles from which we start. Note that at each recursion step, the number of poles to be embedded decreases. If the amount of poles has come down to the size of a single Superpole, we just have to use one Superpole as the recursion anchor. Finally, after having connected all of these skeletons correctly, we may delete all unnecessary parts of our EUG, i.e, those that have no possible path to any input or output of any pole.

We formalize the construction by the definition of EUGs we outlined in Definition 2.0.5, as well as informally prove its correctness.

Definition 3.1.1. *An Augmented k-way Valiant Block* (V, E) *with poles* P*, input nodes* I *and output nodes* O*, to be embedded into a super block, needs to fullfill:*

1. $V = P \cup I \cup O$, $P \cap I = P \cap O = \emptyset$, $|I| = |O| = k$.
2. *Under Valiant's construction especially* $I \cap O = \emptyset$ *holds.*
3. $\Delta(G[P]) = \Delta((P, E^P)) = 2$ *,i.e., the fanin and fanout of all nodes is atmost* 1.
4. $E = E^P \cup E^{io}$*, such that:*
   a) $\forall e \in E^{io} : (e = (\text{in}, p) \wedge in \in I \wedge p \in P) \vee (e = (p, \text{out}) \wedge p \in P \wedge \text{out} \in O)$.
   b) $\forall p \in P : \exists^{\leq 1} in \in I : (\text{in}, p) \in E^{io}$.
   c) $\forall p \in P : \exists^{\leq 1} out \in O : (p, \text{out}) \in E^{io}$.

Now we also define the Superpoles in which we want to embed these Valiant blocks.

Definition 3.1.2. *A k-way* ***Superpole*** $G = (V, E, P, P', I, O)$ *has the following properties:*

1. $P = P' \cup I \cup O$ $P \cap I = \emptyset$ *and* $P \cap O = \emptyset$.
2. $\forall G' = (V', E') \in B_k(P', I, O) : \forall e = (u, v), e' = (u', v') \in E' : \exists E_1, E_2 \subseteq E : isPath(u, v, E_1) \wedge isPath(u', v', E_2) \wedge E_1 \cap E_2 = \emptyset$.

i.e., Superpoles can embed any Augmented Block with edge disjoint paths, while preserving the inputs and outputs of the Block.

The most prominent example for a Superpole is the 2-way Superpole for Valiant's construction as in Fig. 3.1.

The graph of Fig. 3.2 shows how the main skeleton and methodology for the recursion of sub-EUGs are laid out. The algorithm for creating the Valiant EUG is given in Algorithm 3.1 and edge embedding of a graph may be performed by Algorithm 3.2.

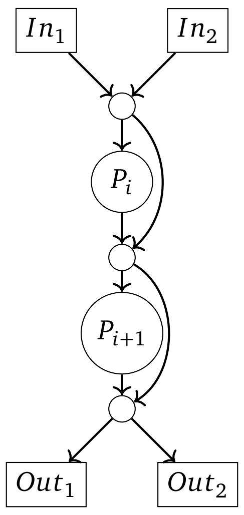

Figure 3.1: Example of a Valiant 2-way Superpole

Algorithm 3.1 *CreateValiantEUG*(P, k)

```
Input:  $P$ pole nodes
        $k$ way-ness of EUG
Output: $\Gamma_1(n)$ EUG $G = (V, E, P, G^*, \{G^i | 1 \le i \le k\})$
1: $V \leftarrow \emptyset$
2: $E \leftarrow \emptyset$
3: $G^* \leftarrow \emptyset$
4: $O^{s^0} \leftarrow$ k dummy nodes
5: for $i \leftarrow 1$ to $\lceil \frac{n}{k} \rceil$ do
6:     $P^{s^i} \leftarrow \{p_{k(i-1)+1,\dots,p_{ki}}\}$
7:     $s^i = (V^{s^i}, E^{s^i}, P^{s^i}, P^{s^i}, I^{s^i}, O^{s^i}) \leftarrow \mathit{InstantiateSuperpole}(P^{s^i}, k)$
8:     $G^* \leftarrow G^* \cup \{s^i\}$
9:     $V \leftarrow V \cup V^i, E \leftarrow E \cup E^i$
10: end for
11: for $i \leftarrow 1$ to k do
12:     if $n \le k$ then
13:         $G^i \leftarrow (\emptyset, .., \emptyset)$
14:     else
15:         $P^i \leftarrow \{O^{s^1}[i], ..., O^{s^{\lceil \frac{n}{k} \rceil - 1}}[i]\}$
16:         $G^i(V^i, E^i, ...) \leftarrow \mathit{CreateValiantEUG}(P^i, k)$
17:         $V \leftarrow V \cup V^i, E \leftarrow E \cup E^i$
18:     end if
19: end for
20: return $G = (V, E, P, G^*, G^1, ..., G^k)$
```

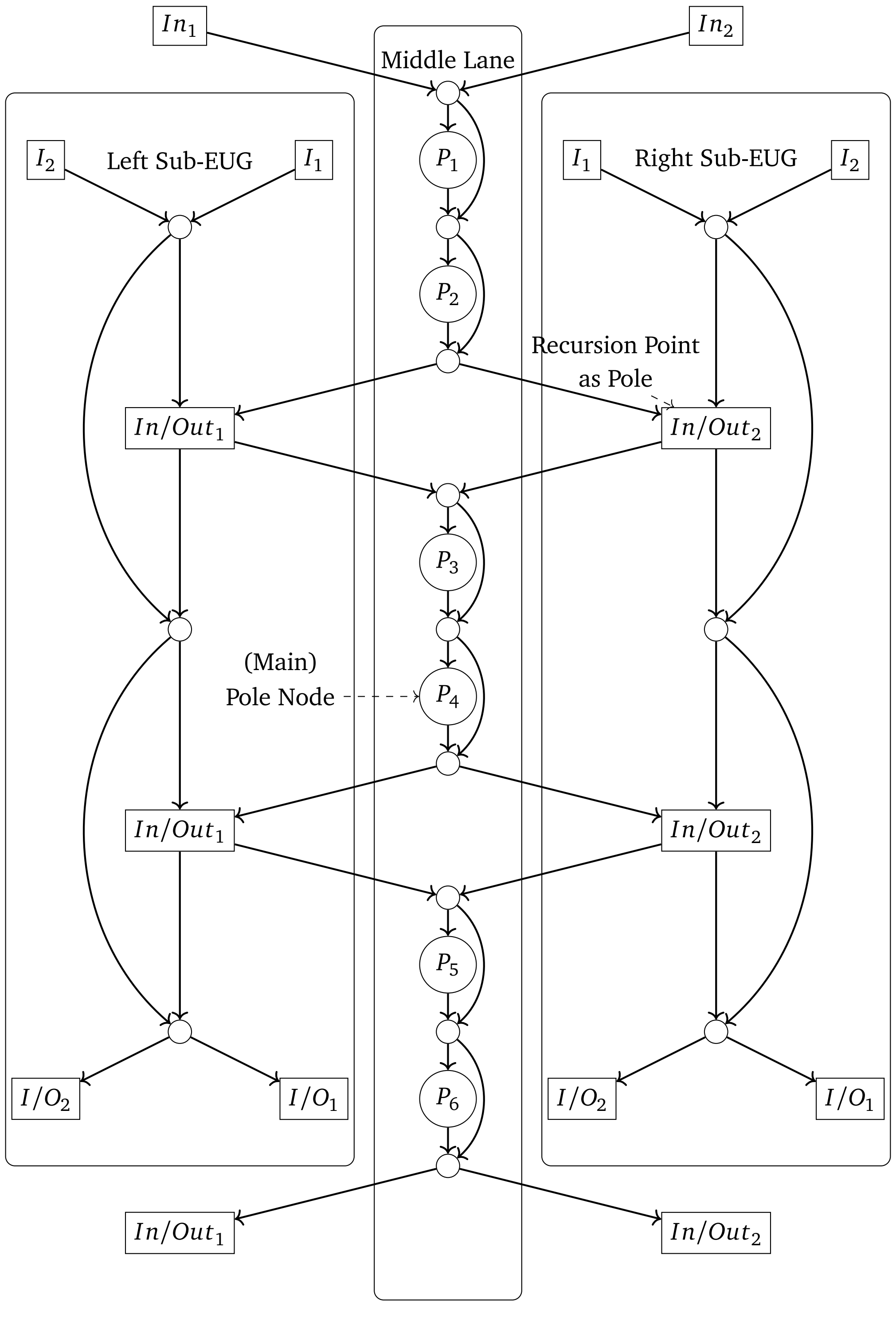

Figure 3.2: Example of a 2-way Valiant EUG with 6 poles annotated to describe the most relevant features

Algorithm 3.2 *EdgeEmbedding*(G, G')

```
    Input:  Valiant EUG $G = (V, E, P, G^* = \{s^1, \dots, s^{\lceil \frac{n}{k} \rceil}\}, G^1, \dots, G^k)$
            Graph to be embedded $G' = (P, E') \in \Gamma_1(n)$
    Output: Edge Embedding map $\varphi : E' \to \mathcal{P}(E)$
 1: if $V = \emptyset$ then
 2:     return $\emptyset$
 3: end if
 4: $A^1, \dots, A^{\lceil \frac{n}{k} \rceil}, R^1, \dots, R^k \leftarrow PathFinding(G, G')$
 5: for $i \leftarrow 1$ to $\lceil \frac{n}{k} \rceil$ do
 6:     $\varphi^{A^i} \leftarrow SuperpoleEdgeEmbedding(s^i, A^i)$
 7: end for
 8: for $j \leftarrow 1$ to $k$ do
 9:     $\varphi^{R^j} \leftarrow EdgeEmbedding(G^j, R^j)$
10: end for
11: $\varphi \leftarrow CombineEdgeEmbeddings(G, G', \varphi^{Ai}, \varphi^{Ri})$
    ▷ Stich all paths together, which overlap Superpole boundaries and copy in mappings
    that stay inside of Superpoles, from the relevant Augmented Block
12: return $\varphi$
```

Note that this algorithm requires the subroutine given in Algorithm 3.3 to find the paths for inside of Superpoles and between them by describing the Augmented Blocks as well as the recursion graphs connecting the Superpoles. An intuitive algorithm for *SuperpoleEdgeEmbedding* can be given for the finite descriptions of Superpoles by simple enumeration of all possible Augmented Blocks.

The correctness of the construction and embedding are proven by Theorem 3.1.1.

Theorem 3.1.1. *CreateValiantEUG*(P, k) *(cf. Algorithm 3.1) produces a k-way 1-EUG for which EdgeEmbedding*(G, G') *(cf. Algorithm 3.1) provides a mapping* φ *that correctly embeds any* $G \in \Gamma_1(|P|)$ *into the EUG.*

Proof. We argue the correctness of the algorithm line by line. First the path finding algorithm *PathFinding*(G, G') is invoked. It determines for each of the Superpoles the Augmented Valiant Blocks A^i to be embedded into Superpoles i, as well as for each of the k sub-EUGs a graph, which has to be embedded into the sub graph. Finally every Superpole is embedded according to its A^i and the embedding is recursively repeated on every sub-EUG j with its graph R^j. Since eventually there are no more crossings needed between sub-EUGs (and there are no more sub-EUGs), recursion anchors will always just be embeddings of the lowest singular Superpoles.

PathFinding(G, G') instantiates an abstract representation of the recursive embedding task on the next layer, called a *recursion graph* $R^\sim$ which will contain all edges that cross between Superpoles. This is done by creating for every Superpole an abstract representation of

Algorithm 3.3 *PathFinding*(G, G')

```
      Input:  Valiant EUG $G = (V, E, P, G^* = \{s^1, ..., s^{\lceil \frac{n}{k} \rceil}\}$
              Graph to be embedded $G' = (P, E') \in \Gamma_1(n)$
     Output:  Augmented Valiant Blocks $A^i$, sub graphs $R^i \in \Gamma(\lceil \frac{n}{k} \rceil - 1)$
 1: for $i \leftarrow 1$ to $\lceil \frac{n}{k} \rceil$ do
 2:     $A^i = (V^{A^i}, E^{A^i}) = (I^i \cup O^i \cup P^i, \emptyset)$
 3: end for
 4: $\tilde{V}^R \leftarrow \{in^i | 1 \leq i \leq \lceil \frac{n}{k} \rceil\} \cup \{out^i | 0 \leq i \leq \lceil \frac{n}{k} \rceil - 1\}$
 5: $\tilde{E}^R \leftarrow \emptyset$
 6: $\tilde{R} \leftarrow (\tilde{V}^R, \tilde{E}^R)$
 7: for $e = (u, v) \in E^{'}$ do
 8:     $s_i \leftarrow$ Superpole of $u$
 9:     $s_j \leftarrow$ Superpole of $v$
10:     if $i \neq j$ then
11:         $\tilde{E}^R \leftarrow \tilde{E}^R \cup \{(out^i, in^j)\}$
12:     end if
13: end for
14: $\{\tilde{R}^i = (\tilde{V}^R, \tilde{E}^i_R)\} \leftarrow k - edge - color(\tilde{R})$
15: for $l \leftarrow 1$ to $k$ do
16:     $IO_l \leftarrow$ Poles of $G^l$
17:     $R^l = (V^l_R, E^l_R) \leftarrow (IO_l, \emptyset)$
18:     for $e = (out^i, in^j) \in \tilde{E}^l_R$ do
19:         $E^l_R \leftarrow E^l_R \cup \{(out^i_l, in^j_l)\}$
20:     end for
21: end for
22: for $e = (u, v) \in E^{'}$ do
23:     $s_i \leftarrow$ Superpole of $u$
24:     $s_j \leftarrow$ Superpole of $v$
25:     if $i = j$ then
26:         $E^{A^i} \leftarrow E^{A^i} \cup \{(u, v)\}$
27:         Get unused $(out^i_x, in^j_x) \in R^x$
28:         $E^{A^i} \leftarrow E^{A^i} \cup \{(u, out^i_x)\}$
29:         $E^{A^j} \leftarrow E^{A^j} \cup \{(in^i_x, v)\}$
30:     end if
31: end for
32: return all $(A^i, R^i)$
```

its interface in $R^{\sim}$ by adding a representative input and output node (the first and final Superpole are missing the input and output node, respectively). For every edge (u, v) in G', *PathFinding*(G, G') now determines if the nodes u and v are members of the same Superpole. If not, it adds an edge into $R^{\sim}$ between the representative interface nodes of the two different

Superpoles i and j. In the end $R^{\sim}$ must be a k colorable graph, because every pair of Superpoles has k poles with in/outdegree 1, thus every two Superpoles may at most have k edges between them. Thus, the algorithm may now edge-color the graph $R^{\sim}$ with k colors. Finally, every R^l graph for the l-th sub-EUG is given, by picking the l-th output node of the i Superpole, as well as l-th input nodes out of the j-th Superpole and adding the edge between them to R^j. (Since all the j-th input/output nodes over all Superpoles on the current recursion step are the poles of the sub-EUG, they are the vertices of R^j).

Now, in order to determine the edge embeddings for each of the super poles, we once again consider all edges in E'. If the edge stays within the same Superpole i, (u,v) is added to A^i. If not (u in i and v in $j \neq i$), we already earlier determined the input/output nodes towards which each edge is supposed to be headed. We now pick the first free l such that a pair (out_l^i, in_l^j) from R^j is still available, and add (u, out_l^i) to A^i as well as (in_l^j, v) to A^j. (By the specification of the sub graphs R^j there will always be an edge available).

□

The overall size (number of nodes) of the EUG is bounded by $\mathcal{O}(n \log n)$ asymptotic complexity. Note that a lower complexity is generally desired, as it allows us to scale our approach in the MPC setting for larger circuits, and therefore larger functions.

Theorem 3.1.2. *The size of a Valiant EUG as given by Algorithm 3.1, is bounded by:*

$$\frac{|Superpole_k|}{k \log_2(k)} n \log_2(n) + \mathcal{O}(n).$$

Proof. First we extract from the algorithm that the recursive relation in the size is bounded by:

$$|Val_k(n)| \leq \lceil \frac{n}{k} \rceil |Superpole_k| + k|Val_k(\lceil \frac{n}{k} \rceil - 1)|.$$

(As some nodes and edges will be removed as optimization, without having an effect on the complexity).

We now prove the final bound by expansion of the recursive term. If the number of poles satisfies $|P| \leq k$, then it holds that $|EUG| \leq |Val(k)| = |Superpole_k|$ as a single Superpole is sufficient to represent those poles.

We may now expand the EUG size recursively as follows:

$$|EUG| \leq |Val_k(n)| \leq \lceil \frac{n}{k} \rceil |Superpole_k| + k|Val_k(\lceil \frac{n}{k} \rceil - 1)|.$$

We evaluate

$$|Val_k(\frac{n}{k})| \leq \lceil \frac{\frac{n}{k}}{k} \rceil |Superpole_k| + k|Val_k(\lceil \frac{\frac{n}{k}}{k} \rceil - 1)|$$

$$= \lceil \frac{n}{k^2} \rceil |Superpole_k| + k|Val_k(\lceil \frac{n}{k^2} \rceil - 1)|.$$

If we round up the ceiling terms, we get a simpler bound

$$\leq (\frac{n}{k^2} + 1)|Superpole_k| + k|Val_k((\frac{n}{k^2} + 1) - 1)|.$$

By expanding the term repeatedly, the sum is extended, and we can bound the recursive term of the size:

$$|EUG| \leq |Val(n)| \leq \Sigma_{i=0}^{h-1}((\frac{n}{k} + k^i) * |Superpole_k|) + k^h |Val_k(\frac{n}{k^h})|$$

where h is the depth of the recursion, i.e, the amount of Valiant recursions we need until the sub-EUG to be built only needs to accommodate $\leq k$ poles, $h = \min\{i \in \mathcal{N} | n \leq k^{i+1}\}$.

Since as argued previously the last step has $|Val_k(\frac{n}{k^h})| \leq |Superpole_k|$, and thus

$$\begin{aligned}
|Val(n)| &\leq \Sigma_{i=0}^{h-1}((\frac{n}{k} + k^i) * |Superpole_k|) + k^h |Superpole_k| \\
&= (\Sigma_{i=0}^{h-1}(\frac{n}{k} + k^i) + k^h)|Superpole_k| \\
&= (\Sigma_{i=0}^{h-1}(\frac{n}{k}) + \Sigma_{i=0}^{h-1} k^i + k^h)|Superpole_k| \\
&= ((\frac{h * n}{k}) + \Sigma_{i=0}^{h} k^i)|Superpole_k| \\
&= (\frac{hn}{k} + \frac{k^{h+1} - 1}{k - 1})|Superpole_k|
\end{aligned}$$

By definition we have $h = \lceil \log_k(n) - 1 \rceil$ and thus finally

$$\begin{aligned}
Val_k(n) &\leq \lceil \log_k(n) - 1 \rceil \frac{n}{k} |Superpole_k| + \frac{k^{\lceil \log_k(n) - 1 \rceil + 1} - 1}{k - 1} |Superpole_k| \\
&\leq \log_k(n) \frac{n}{k} |Superpole_k| + \frac{k^{log_k(n) - 1 + 1} - 1}{k - 1} |Superpole_k| \\
&= \log_k(n) \frac{n}{k} |Superpole_k| + \frac{kn - 1}{k - 1} |Superpole_k| \\
&= \frac{\log_2(n)}{\log_2(k)} \frac{n}{k} |Superpole_k| + \frac{nk - 1}{k - 1} |Superpole_k|
\end{aligned}$$

The linear term is then given by:

$$\frac{nk - 1}{k - 1} |Superpole_k| \in \mathcal{O}(n).$$

$\square$

3.2 Liu et al's (2-Way) Construction [LYZ+21]

Liu et al's [LYZ+21] EUG construction yields an asymptotic improvement of $3n \log n$ compared to the previous $4.5n \log n$ by [ZYZL19].

The construction starts by constructing a so called *Weak* Liu$^+$ *EUG*. This first weak EUG is still an EUG because it can edge embed every graph, but it is not yet better in size (in fact it is actually worse), as well as not acylic, making it unusable for MPC. The EUG is then however improved by a procedure called *Shortening*.

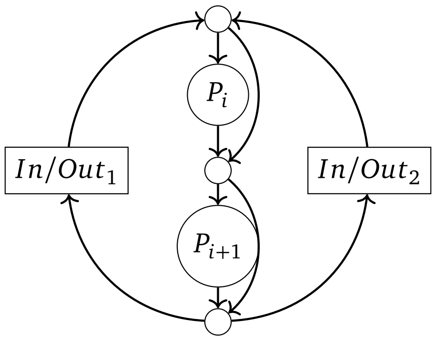

Figure 3.3: Example of a 2-way Liu Superpole

Shortening removes the cycles of the graph by removing the back-looping recursion nodes as well as providing alternative paths from the poles to the sub-EUGs, thus practically removing the recursion nodes on every recursion step .The first change of the construction lies in the change of the Superpoles. A Liu$^+$ k Superpole is given by merging the in/out recursion points of Valiant EUGs that are supposed for the same sub-EUG. For example the 2-Way Liu et al's EUG obtained from the 2-Way Valiant EUG may look like Fig. 3.3.

The resulting Superpoles are once again instantiated in multiples, i.e, $\lceil \frac{n}{k} \rceil$ Liu$^+$ Superpoles are created for a graph with n poles to be embedded. This time however, the Superpoles are not connected to the input/output recursion points of their neighboring Superpoles but are simply arranged in an array. This will now cause us to need more Superpoles in the sub-EUGs, more specifically exactly one more per sub-EUG (for every k sequent Superpoles we add one Superpole to the sub-EUG). This is the main cause of the increased complexity in the weak construction as the first input and last output recursion points which are now also output/input recursion points, are not left out of the recursion. An example of the new recursive structure can be seen in Fig. 3.4.

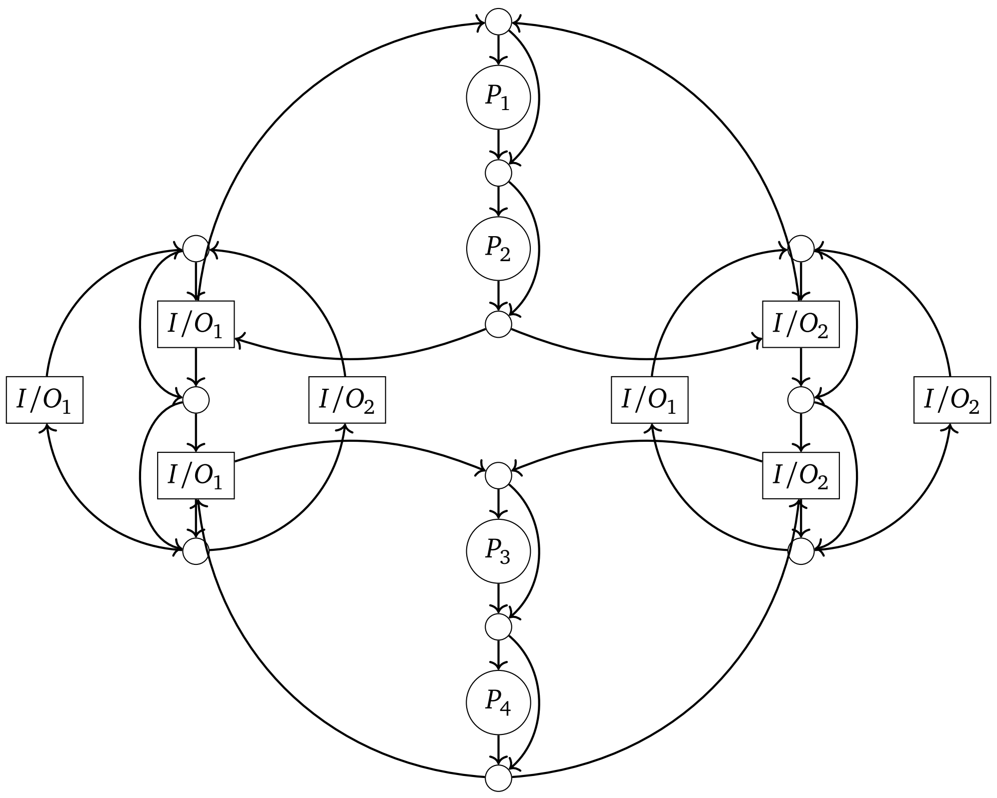

Figure 3.4: Example of a full Weak Liu$^+$ EUG with 4 poles

Formally the construction algorithm for Weak Liu$^+$ EUGs is given by Algorithm 3.4.

The given Weak Liu$^+$ Graph is also an EUG (except for the fact that it has cycles, which we choose to ignore for now) for the same reason as the Valiant EUG (the proof is almost exactly the same, with slight differences in the way we choose the recursion points for embeddings), because both use the principal of interfacing recursion points. Superpoles internally can represent any Γ_1 Graph with the amount of poles inside the Superpole while still allowing arbitrary interfacing with recursion points. The recursion points still act as ports to sub-EUGs for the poles in the EUG to solve the recursive embedding algorithm. The difference, that the recursion points produce acyclic elements makes no difference, as the same quantity of interfacing in/outgoing edges is provided per sub-EUGs, while not having cycles in the embeddings (since any embedding with a cycle must head to a pole that is earlier in the topology).

The overall size of the EUG is again bounded by $\mathcal{O}(n \log n)$ asymptotic complexity again. We prove this once again by the same method as [Wig21].

Algorithm 3.4 *CreateWeakLiuEUG*(P,k)

```
    Input:  $P$ pole nodes
            $k$ way-ness of EUG
    Output: $\Gamma_1(n)$ EUG $G = (V, E, P, G^*, \{G^i | 1 \leq i \leq k\})$
 1: $V \leftarrow \emptyset$
 2: $E \leftarrow \emptyset$
 3: $G^* \leftarrow \emptyset$
 4: $O^{s^0} \leftarrow$ k dummy nodes
 5: for $i = 1 to \lceil \frac{n}{k} \rceil$ do
 6:     $P^{s^i} \leftarrow \{p_{k(i-1)+1,\dots,p_{ki}}\}$
 7:     $s^i = (V^{s^i}, E^{s^i}, P^{s^i}, P^{s^i}, I^{s^i}, O^{s^i}) \leftarrow InstantiateSuperpole(P^{s^i}, k)$
 8:     $G^* \leftarrow G^* \cup \{s^i\}$
 9:     $V \leftarrow V \cup V^i, E \leftarrow E \cup E^i$
10: end for
11: for $i \leftarrow 1$ to k do
12:     if $n \leq k$ then
13:         $G^i \leftarrow (\emptyset, .., \emptyset)$
14:     else
15:         $P^i \leftarrow \{O^{s^1}[i], \dots, O^{s^{\lceil \frac{n}{k} \rceil}}[i]\}$
16:         $G^i(V^i, E^i, \dots) \leftarrow CreateWeakLiuEUG(P^i, k)$
17:         $V \leftarrow V \cup V^i, E \leftarrow E \cup E^i$
18:     end if
19: end for
20: return $G = (V, E, P, G^*, G^1, \dots, G^k)$
```

Theorem 3.2.1. *The size of a Weak* Liu$^+$ *EUG as given by Algorithm 3.4, is bounded by:*

$$\frac{|Superpole_k|}{k \log_2(k)} n \log_2(n) + \mathcal{O}(n)$$

Proof. First, we extract from the algorithm that the recursive relation in the size is bounded by:

$$|WeakLiu_k(n,i)| \leq \lceil \frac{n}{k^{i+1}} \rceil |Superpole_k| + k|WeakLiu_k(\lceil \frac{n}{k} \rceil, i+1)|$$

where i is the current height index of the recursion (we start with $i = 0$ and end with the total depth of the recursion tree $i = h = \lceil \log_k(n) - 1 \rceil$).

For $n \leq k$, we have $WeakLiu_k(n,i) \leq |Superpole_k|$. Note that we can bound the amount of Superpoles on the i-th recursion step by $\lceil \frac{n}{k^{i+1}} \rceil$ because the original n pole nodes are partitioned k way into Superpoles (leading to the one extra partition) and on every subsequent recursion step (where the amount of poles of one sub-EUG is equal to the amount of Superpoles in the previous recursion layer) it is once again partitioned k way.

Note also that this implies that the end of the recursion is at $i = h$ as the amount of Superpoles needed is

$$\begin{aligned}\lceil \frac{n}{k^{h+1}} \rceil &\leq \lceil \frac{n}{k^{\lceil \log_k(n)-1 \rceil + 1}} \rceil \\ &\leq \lceil \frac{n}{k^{\lceil \log_k(n) \rceil - 1 + 1}} \rceil \\ &= \lceil \frac{n}{k^{\lceil \log_k(n) \rceil}} \rceil \\ &\leq \lceil \frac{n}{n} \rceil \leq 1\end{aligned}$$

which implies that the amount of poles on that recursion step are the first to be smaller than k (as the amount of poles per recursion step is a monotonically decreasing function). If we now expand the sum we get

$$\begin{aligned}|WeakLiu_k(G)| \leq |WeakLiu_k(n,0)| &\leq \Sigma_{i=0}^{h-1} k^i \lceil \frac{n}{k^{i+i}} \rceil |Superpole_k| + k^h |Superpole_k| \\ &\leq \Sigma_{i=0}^{h-1} k^i (\frac{n}{k^{i+1}} + 1) |Superpole_k| + k^h |Superpole_k| \\ &\leq \Sigma_{i=0}^{h-1} (\frac{n}{k} + k^i) |Superpole_k| + k^h |Superpole_k|\end{aligned}$$

$$\begin{aligned}&= \Sigma_{i=0}^{h-1} \frac{n}{k} |Superpole_k| + \Sigma_{i=0}^{h-1} k^i |Superpole_k| + k^h |Superpole_k| \\ &= (\frac{k^{h+1}-1}{k-1} + \frac{hn}{k}) |Superpole_k|.\end{aligned}$$

Emplacing the definition of h finally yields

$$\begin{aligned}|WeakLiu_k(G)| \leq WeakLiu_k(n,0) &\leq (\frac{k^{\lceil \log_k(n)-1 \rceil + 1} - 1}{k-1} + \frac{\lceil \log_k(n) - 1 \rceil n}{k}) |Superpole_k| \\ &\leq \frac{\log_2(n)}{\log_2 k} \frac{n}{k} |Superpole_k| + \frac{k^{\lceil \log_k(n)-1 \rceil + 1} - 1}{k-1} |Superpole_k| \\ &\leq \frac{\log_2(n)}{\log_2 k} \frac{n}{k} |Superpole_k| + \frac{k^{\lceil \log_k(n) \rceil - 1 + 1} - 1}{k-1} |Superpole_k| \\ &\leq \frac{\log_2(n)}{\log_2 k} \frac{n}{k} |Superpole_k| + \frac{k^{\log_k(n)+1} - 1}{k-1} |Superpole_k| \\ &\leq \frac{\log_2(n)}{\log_2 k} \frac{n}{k} |Superpole_k| + \frac{kn-1}{k-1} |Superpole_k|.\end{aligned}$$

The second term of the sum once again yields the linear remainder term to the complexity. □

Note that this Weak Liu construction so far has a significantly worse performance than the regular Valiant construction, but opens up the new road of circuit shorting for optimization. We now introduce the shorting procedure (cf. Algorithm 3.5), which turns the EUG into a DAG and decreases the size significantly.

Algorithm 3.5 *ShortWeakLiu*(G)

```
    Input:  $\Gamma_1(n)$ weak Liu EUG $G = (V, E, P, G^*, \{G^i | 1 \leq i \leq k\})$
   Output:  $\Gamma_1(n)$ Liu EUG $G = (V, E, P, G^*, \{G^i | 1 \leq i \leq k\})$
 1: if $V = \emptyset$ then
 2:     return $(\emptyset, \emptyset, \ldots)$
 3: end if
 4: for $(u, v) \in E$ do
 5:     if $u \in s$ and $v$ is recursion point for some Superpole $s \in G^*$ then
 6:         $G^x \leftarrow$ EUG of the pole $v$
 7:         $E \leftarrow E - \{(u, v)\}$
 8:         $W \leftarrow \Gamma^-_{G^x}(v)$
 9:         $E \leftarrow E - \{(v, w) | w \in W\}$
10:         $E \leftarrow E \cup \{(u, w) | w \in W\}$
11:     else if $u$ is recursion point for some Superpole $s^*$ and $v \in s$ then
12:         $G^x \leftarrow$ EUG of the pole $u$
13:         $E \leftarrow E - \{(u, v)\}$
14:         $W \leftarrow \Gamma^+_{G^x}(v)$
15:         $E \leftarrow E - \{(w, u) | w \in W\}$
16:         $E \leftarrow E \cup \{(w, v) | w \in W\}$
17:     end if
18:     Remove recursion points from $G$
19:     for $i = 1$ to $k$ do
20:         $G^i \leftarrow ShortWeakLiu(G^i)$
21:     end for
22: end for
23: return $G$
```

Essentially, Algorithm 3.5 eliminates the recursion points on all sub-EUGs recursively and reroutes the edges which would interface through them. Any Edge heading into the recursion point of a Superpole, which was always from a sub-EUG by the design of the Superpoles, would now instead head to the successor of that recursion point in the same Superpole. Any edge starting in one of the recursion points and going into one of the nodes of the Superpole, in which the recursion point was a pole node, would instead start from the predecessor of the recursion point inside of its Superpole. Edges, of the Superpoles, which did not cross the border from Superpole to sub-EUG would simply be removed.

We now want to argue why this shorting process yields a significantly smaller, acylcic EUG.

Theorem 3.2.2. *Applying Algorithm 3.5 to a Weak Liu EUG results in an acylic EUG.*

Proof. If we start from the edge embeddings of two graphs $G, G' \in \Gamma_1(n)$ into the original Weak Liu EUG, we now want to argue, that we can modify their embedding to fit into the new shorted graph. The mapping of the poles remains simply the same, as all the poles on the middle lane of the Weak Liu EUG still exist. We now only need to show, that the edge embedding can be rerouted/ shorted, to accommodate the fact, that we now no longer have recursion points. If the embedded edge stays inside of the same Superpole, this does not require any change, as the paths inside of the middle lane Superpoles are unaffected. If we consider now a path to a different Superpole, it would have headed through at least two recursion points, one as an ingoing interface into a sub-EUG and one as an outgoing choice. For the recursion point serving as an input, we know that the edge of the path going into the recursion point must have been rerouted to the same node as the one which is the next node in the path after the recursion point, because the edge embedding from the recursion point can only head to a subsequent recursion point in the sub-EUG by that edge (otherwise it would have headed back into the Superpole to another pole through the edge that was removed). Thus we can replace the two edges from the original embedding, the ingoing into the recursion point and $a = (u, r)$ the outgoing $b = (r, v)$, by the singular edge that was added in the Shortening directly (u, v). An analogous replacement can be chosen for the output recursion point. Because every two removed edges have exactly one unique new edge and this segment of each disjoint embedding is thus replaced with a unique new edge per path, the new embedding must also be edge disjoint. Since any two graphs G, G' to be embedded could have been chosen, the embedding stays universal.

The EUG is a DAG as all the edges which produced loops (which headed through the recursion points) have been removed and instead been replaced by edges that are lower in the topological order. □

Now the complexity of the EUG size has also slightly decreased, which yields the main improvement of Liu's construction.

Theorem 3.2.3. *Applying the Shortening Algorithm 3.5 to a EUG produced by Algorithm 3.4, the resulting EUG has a total size bound by:*

$$|ShortWeakLiu(WeakLiu(G))| \leq \frac{|Superpole_k| - k}{k \log_2(k)} n \log_2(n) + \mathcal{O}(n)$$

Proof. The construction is given by removing every recursion point in every Superpole of every sub-EUG (inclusive the original total EUG).

Note that the amount of Superpoles can be found by taking the complexity of the Weak Liu construction and divide it by the size of the Superpoles.

$$\frac{|WeakLiu(G)|}{|Superpole_k|} \geq \frac{n \log_2(n)}{k \log_2(k)} + \mathcal{O}(n).$$

Since we now remove k Superpoles per each Superpole, we get as a final complexity

$$\begin{aligned}|ShortWeakLiu(WeakLiu(G))| &\leq |WeakLiu(G)| - k\frac{|WeakLiu(G)|}{|Superpole_k|}\\ &\leq \frac{|Superpole_k| - k}{k\log_2(k)} n\log_2(n) + \mathcal{O}(n).\end{aligned}$$

□

As [LYZ+21] points out, and can be seen by some consideration of the size of Superpoles, choosing $k = 2$ yields the optimal size construction for this framework with the more precise size complexity of $\approx 3n\log_2(n) + \mathcal{O}(n)$, which is as of time of this writing, the best known practical implementation of EUGs.

3.3 Miscellaneous Constructions

There exist several other constructions for UCs besides the original framework for construction, that was proposed in [Val76]. [ZYZL19] optimized the construction for a 4-way in Valiant's classical construction, by optimizing the Superpoles with 4 poles. [AGKS20] introduced a hybrid construction of 2- and 4-way Valiant Recursion, improving the complexity bound from $4.75n\log n$ to $4.5n\log n$. This is achieved by adaptively choosing the way-ness parameter in the recursion steps depending on what best reduces size.

[SS09] provides an overview over several constructions which are useful in a setting of multi-input gate circuits, especially for smaller scale inputs circuits, by combining special functional blocks of permutation and computation iteratively, instead of recursively embedding the topology of the circuit in sub graphs. Their first iterative construction has complexity $\mathcal{O}(n^2\log d)$. Their second construction uses three types of modules (input selection, output selection and a generalized universal blocks) and achieves complexity $\mathcal{O}(dn\log^2 n)$. The third construction uses the recursive nature of [Val76] and modifies a circuit to have a fanin and fanout of at most d with some overhead. The first two constructions are more beneficial for smaller circuits to be embedded, while the third construction may be used for larger circuits, and is related to our work in the sense that in it a circuits fanout is reduced before passing it to a d-way generalization of a classical construction like Valiant's.

It should be noted, that one of the main targets of all of these constructions is to decrease the amount of AND gates inside of the circuit (which mostly overlaps with the goal of minimizing overall size) in order to decrease the amount and duration of communication necessitated for MPC evaluation (only AND gates need to create fully randomized garbled tables per gate, which need to be transmitted as data in the protocol, XOR gates are cheaper to be evaluated, as the XOR gate is evaluated via XOR operations on the protocol level without extra space/ communication requirements).

3.4 Translating an EUG into a Universal Circuit

Before we can finally build Universal Circuits from EUGs, we define what precisely a Universal Circuit should be.

Definition 3.4.1. *Let $u+g+v=n$, $\mathbb{B}$ be the set of all Combinatorial Boolean Functions, $\mathbb{B}(u,g,v)$ be the restriction of $\mathbb{B}$ to circuits with u inputs, g gates and v outputs, and $h \in \mathbb{N} \to \mathbb{N}$ an arbitrary function that maps the size of the circuit to the size of the* programming p_f.
*Then we define a Universal Circuit (**UC**) as a Combinatorial Boolean Circuit*
$UC : \{0,1\}^u \times \{0,1\}^{h(g)} \to \{0,1\}^v$*, constructed by some deterministic algorithm*
$CreateUC : \mathbb{B}(u,g,v) \to \mathbb{B} \times \{0,1\}^{h(n)}$*, that fulfills the following properties:*

1. *(Correctness)* $\forall f \in \mathbb{B}(u,g,v) : \forall x \in \{0,1\}^u : \exists p_f \in \{0,1\}^{h(n)} : (UC, p_f) = CreateUC(f) \wedge UC(x, p_f) = f(x)$
2. *(Function Hiding)* $\forall f, g \in \mathbb{B}(u,g,v) : (UC_f, p_f) = CreateUC(f) \wedge (UC_g, p_g) = CreateUC(g) \wedge UC_f = UC_g$

It should be noted that this definition is only exact for true PFE as it denotes the exact hiding property up to the size information, whereas for SPFE, we would slightly modify it to lessen the function hiding property to e.g. a statistical or computational hiding property, or by adjusting the relevant function spaces, such that only functions with more similar attributes have the same UC. E.g. [BHR12] defines function hiding in terms of *obliviousness* of garbling schemes, by defining a game in which any polynomial probabilistic Turing machine Adversary can only learn some meta-information (f) of the function with negligible probability. They further define $_{size}(f) = \{u, g, v\}$ as the meta-information function, if Universal Circuits are used. Instead choosing $(f) = \{u, g, v, \Delta(Graph(f))\}$ might lead to a useful definition for multi input-multi output Universal Circuits in our fixed construction.

As outlined in [AGKS20], we must now perform some slight modifications to the EUG to allow for the computation in a Boolean setting. We will explain the process for the classical 2-way setting of [LYZ+21] and [Val76], where there only exists gates with 2 inputs and one output (which may be used twice). The adaption towards a setting with more inputs (and outputs) will be explained in § 4.

Assume first we are given a circuit C built only from binary gates with a total of n gates. Note that C has a natural graph representation $Graph(C) = G = (V, E)$ that preserves the topology of the circuit, the wiring by edges in E, and gates by nodes in V. We now perform the construction and embedding as laid out for either Valiant[Val76] or Liu et al. [LYZ+21] with the given graph G and are given an acyclic 2 EUG (i.e, one that embeds $\Gamma_2(n)$) and an embedding mapping φ which embeds the graph into that EUG.

The transformation into a UC introduces three new types of special-purpose Boolean gates.

X and Y Gates are specific utility gates which serve to hide the paths of the edge embedding. They essentially behave like rail yard switches by choosing which of the inputs is given to

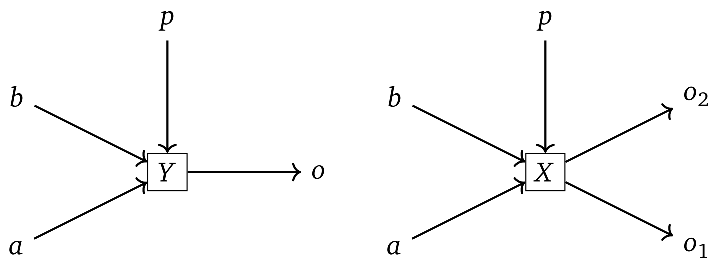

Figure 3.5: X and Y Gate

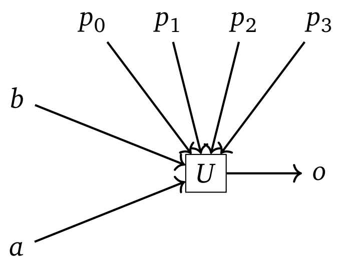

Figure 3.6: Universal Gate

which output. A Y Gate essentially then is a multiplexer that only chooses one of the two inputs and a X Gate switches the two inputs or forwards them unaltered. These two gates with the two possible configurations capture all possible options of embedding that can occur; We enforced in our constructions, that every node, not just poles, has in/outdegree 2 (as can be seen in the recursive wiring to the sub-EUGs as well as in the Superpoles) and the embeddings were edge disjoint over all paths.

Universal Gates are the representations of our original gates, which are programmed by their truth table as programming bits. Universal Gates have, as enforced on C, only 2 inputs and by our design now a single output and can compute any single Boolean gate. The correct topological order of the Universal Gates as well as their connection is enforced by the embedding.

We may thus define the gates by the following Boolean formulae:

X-Gate: $(o_1, o_2) = ((\neg p \wedge a) \vee (p \wedge b), (\neg p \wedge b) \vee (p \wedge a))$

Y-Gate: $o = (\neg p \wedge a) \vee (p \wedge b)$

Universal Gate: $o = Mux(Mux(p_0, p_1, b), Mux(p_2, p_3, b), a)$ s.t. $Mux(x_0, x_1, b) = x_b$

We now translate the EUG into a UC as follows.

1. Every pole on the middle lane that represents a gate in the embedding, is replaced with a Universal Gate (cf. Fig. 3.6), and its programming is derived by taking the truth table from the Gate that is mapped to that pole.

2. Every node or pole in one of the sub-EUGs with 2 inputs and 1 outputs is replaced by a Y-Gate (cf. Fig. 3.5). Its programming bit is derived by how paths are wired through the node in the original EUG by the edge embedding.
3. Every node or pole in one of the sub-EUGs with 2 inputs and 2 outputs is replaced by a X-Gate (cf. Fig. 3.5). Its programming bit is derived by how paths are wired through the node in the original EUG by the edge embedding.
4. Every pole on the middle lane that represents an input or output in the embedding is transformed into that input/output wire.

Note that all of the information that forces the new UC to behave as our original circuit is captured into programming of the X and Y gates as well as truth tables. Also note that any two different circuits with the same amount of total gates as well as the same number of inputs and outputs get translated into the same UC but with different programming bits. Thus, in the default construction with the same amount of inputs and outputs per pole, the UC only leaks the size of the embedded circuit as well as the amount of inputs, outputs and gates. By now treating that programming information (i.e., the collection of programming bits in the correct order) as a secret in multiparty computation, having them be a secret input of the function holder in the multi party computation, we enforce the privacy of the function in the computation. As the regular input size must be known by the input provider in the computation (and possible output size by the design choice of what output information is given to which party) in order to correctly compute, that information is also leaked. The actual input values stay private however, as they are obfuscated by the computation (the input provider knows which nodes in the universal circuit represent the input to the computation and can give them as secret inputs in the MPC). Thus this construction of a universal circuit from an EUG, together with a secure MPC protocol gives a correct and private procedure for private function evaluation.

4 Design and Implementation

We now describe our novel approach to allow for the multi input/output gates. There are two main ideas. Either we create more copies of the EUG to accommodate a higher fanin/fanout in the graph to be embedded, or we reduce the graph to be embedded to a lower fanin/fanout representation and post-process the graph via an auxiliary graph construction. Most previous graph embeddings where done by reducing the fanin/fanout via copy gates/truth table partition into a graph with fanin/fanout 2 and then embedding it into 2 EUGs. This however is not optimal towards a lower size, as we can instantiate larger Universal Gates (*ULUT*s) cheaper. It should be noted, that our construction requires either Valiant's [Val76] or Liu et al's[LYZ+21] EUG constructions as presented in § 3, in order to construct its EUGs.

4.1 Representation of larger Look Up Tables

In order to really benefit from larger LUTs we must first pin down how we construct them from simple components. This is necessary, as not all MPC protocols support special features for LUTs and must have some lowering of LUTs into Boolean primitives.

Definition 4.1.1. *A $n \rightarrow m$ ULUT is a lookup table with 2^n rows and m output columns, such that each output column has in any row a variable $c_i, 0 \leq i \leq 2^n * m$, which we call the programming bit of the ULUT for some specific inputs $x = (x_0, ..., x_n)$ on the specific output slot.*

*ULUT*s will be used in place of Universal Gates (cf. Fig. 3.6), to embed LUTs directly into the poles of the UC. Because we no longer need to support the embedding of edges between more primitive gates and instead collect them into the *ULUT* (which represents the many primitive gates), we cut down on the size of the UC while only sparingly increasing the size of each Universal Gate representation (we limit the size of each *ULUT* by the same hyper-parameter k, thus limiting the size of it).

4.1.1 *ULUT* by Chaining of Multiplexers

First we want to solve the case for fanin k and fanout 1. The simplest construction for this, is chaining together multiple Multiplexers.

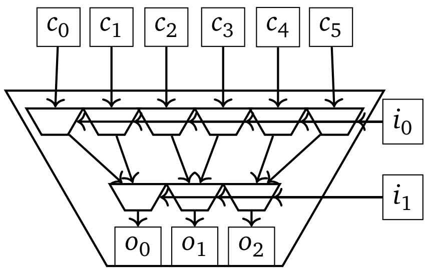

Figure 4.1: Example of a $2 \to 3$ *ULUT*

A $1 \to 1$ *ULUT* can be given by a single multiplexer.

$$ULUT(x, (c_0, c_1)) = Mux(c_0, c_1, x)$$

Where Mux is a Boolean multiplexer with behavior $Mux(c_0, c_1, x) = c_x$, and c_0, c_1, x are the programming bits and the input of the *ULUT* respectively. Now inductively assume we can construct an $n \to 1$ *ULUT* with $(c_0, ..., c_{2^n-1})$ programming bits, then a $n+1 \to 1$ ULUT can be given by

$$\begin{gathered} ULUT((x_0, ..., x_{n+1}), (c_0, ..., c_{2^{n+1}-1})) = \\ Mux(ULUT((x_0, ..., x_n), (c_0, ..., c_{2^n-1})), ULUT((x_0, ..., x_n), (c_{2^n}, ..., c_{2^{n+1}-1})), x_{n+1}) \end{gathered}$$

This technique of chaining Multiplexers is also known as *Shannon Expansion* [Sha49]. It is trivial to see that this recursive construction can represent all functions with the given size, simply because it is a larger lookup table.

Note that this construction size and the amount of programming bits scale exponentially with the number of inputs.

4.1.2 $n \to m$ ULUT

If we need to represent a *ULUT* with multiple different outputs, we simply create m copies of our $n \to 1$ *ULUT* construction and adjust the programming bits of each with the given column of the *ULUT*. This adjustment for multiple different outputs only scales linearly in size with the amount of outputs required. An example of such a *ULUT* can be seen in Fig. 4.1.

Now we have outlined how $ULUT$s could practically be instantiated in an MPC setting, as output of the compiler however we only output the abstract description of each *ULUT*, i.e., input and output size as well as the programming bits for each of the output slots in order.

4.2 Multi Input/Output ABY-UC Format

We now provide a description format for *ULUT* based Universal Circuits in a similar syntax to the classical ABY-UC [DSZ15] format. The circuit is described by two files, the *circuit* and the *programming*. The circuit file has, topologically sorted, one gate of the UC per line, where the first token describes the gate type (X, Y, and U for ULUT) and the subsequent lines describe in/outgoing wires. Every wire is mapped to a natural number and may only be defined once and used an arbitrary amount.

X and Y gates are encoded by the following scheme:

```
Y 1 2 3
X 1 2 3 4
```

The first two numbers are the wire numbers of the inputs, the following numbers are the outputs. If the programming bit is set to 1, the first input is directed to the first output (and the second input on the second output), else the second input is directed to the first output (and the second input to the first output).

*ULUT*s (Universal Look Up Tables) are encoded as follows

```
U 1 2 3 | 4 5
```

the wire numbers ahead of the separator | are the inputs, and those after are the outputs. This separator is necessary, as otherwise, the amount of outputs would be ambiguous. For example in Fig. 4.2 without the separator, we could either have a $4 \rightarrow 1$, $3 \rightarrow 2$ or $2 \rightarrow 3$ *ULUT*. The programming bits are also separated in order to associate them by order to the output wires. The programming bits are to be understood as look up tables, one per output of the *ULUT*. Given the input wires by order, the binary sum of the input values determines the output to be given.

All the programming bits are stored in the separate programming file, where each line contains the programming bits associated to the gate in the same line of the circuit file.

If we evaluate as an example the given circuit file in Fig. 4.2, if the wire 3 holds value 0, 4 holds 1 and 5 holds 1, then the third value (counting from 0) is the correct output, thus 6 would hold 0 and 7 would give out 1.

```
——CIRCUIT——
C 0 1 2
X 1 0 3 4
Y 3 4 5
U 3 4 5 | 6 7
O 6 7
——PROGRAMMING——
0
1
1 1 1 0 0 0 0 1 | 0 1 0 1 1 1 0 0
```

Figure 4.2: Example of an ABY circuit file

4.3 Fixed ρ fanin/fanout Approach

This approach is a slight modification of fixed approach presented by [Wig21] and builds upon the ideas of [Wig21; SS09]. For this approach we recall that each EUG of Valiant or Liu can accommodate an arbitrary graph $G \in \Gamma_1(n)$ (cf. Theorem 3.1.1) with fanin/fanout 1 and n nodes that an arbitrary amount of graphs ρ can be merged to allow for any graph of some fixed maximum fanin/fanout ρ can be handled (cf. Theorem 2.0.3). Thus, if we now have, due to larger *ULUT*s, a graph with higher fanin/fanout, instead of decreasing it using copy gates, we now simply create more copies of the EUG. We now may edge-color the larger fanin/fanout graph with more than two (namely maximum fanin/fanout) colors and embed them into the larger amount of EUGs we created.

This approach also needs one minor adjustment aside from the multiple copies. Since each node (LUT) may now have two semantically differing outputs (because there may be multiple outputs in the LUT with distinct targets), we need some way of distinguishing the value, an outgoing edge receives in the UC representation of the graph.

This is done by assigning each edge in the original graph representation of the circuit so-called slot numbers. One slot number is given to determine the output of the node we are exiting from and another is given to determine the input which this slot feeds into. That extra information on our graph is necessary, as multiple wires (edges) outgoing from one node may share the same slot (an output value might be used in multiple places) and we could not infer this double use just by the order of the edges. This allows us to no longer have to micromanage the order of input/output edges of the graphs in the transformation from EUGs to UC, as we can simply refer to the slot numbers of the edges instead. Note that this slot information is only needed for the creation of the UC from EUG and is not used inside of the EUG construction or the edge embedding. It is only visible in the final UC from the wire numbers associated with each input and output wire. The slot assignment of a circuit to be embedded might look as in Fig. 4.3.

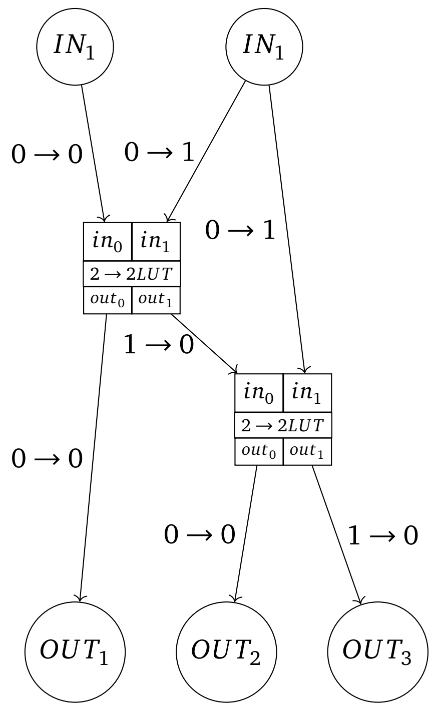

Figure 4.3: Example of DAG to be embedded with the assigned *from* and *to* slot numbers on each edge.

To translate the k-EUG with its edge embeddings into the UC, we consider each edge tuple $(u, from-slot, to-slot, v)$ from our annotated graph. For each of those tuples we use up one of the paths from u to v in the EUG and annotate the slot information on the first and final edge of the path inside of the EUG respectively. The slot information on the edges of the EUG are then used to determine the precise wire number of each input and output. The *ULUT*s are programmed according to their relative *LUT*s. X and Y gates are derived the same way as in the classical constructions (cf. § 3.4). Note that we use slot information for the X and Y Gates as well to simplify the translation (here the numbers are always the same and either 0 or 1). The pipeline for generating a fixed-k multi input/multi output UC may now look as in Algorithm 4.1

4.4 Dynamic/In-Place fanin/fanout Approach

This approach is a slight modification of the dynamic construction approach presented by [Wig21] in order to accommodate *ULUT*s with multiple different numbers of outputs. This adaption again adds the ability to give each LUT in the UC an individual number of outputs wires, but comes at the cost of only having SPFE, as the individual sizes of the LUTs give more information to an adversary, who wants to determine the function that is being embedded.

Algorithm 4.1 *CreateFixedUC*(C, k)

```
Input:  Circuit $C$                                       ▷ Circuit to be embedded
        $k \geq \max\{v \in Graph(C) | \Delta(v)\}$       ▷ Number of mergings
Output: Universal Circuit with programming
$(G'_i)_{i\in[k]} = k - edge - color(G)$
$(G_i)_{i\in[k]} = (\emptyset, \emptyset)^k$
$(\varphi_i)_{i\in[k]} = (\emptyset)^k$
for $i \leq k$ do
    $G_i \leftarrow CreateWeakLiu/ValiantEUG(G_i)$        ▷ (cf. Algorithm 3.4, Algorithm 3.1)
    $\varphi_i \leftarrow EdgeEmbedding(G_i, G'_i)$       ▷ (cf. Algorithm 3.2)
    if algorithm $= Liu$ then
        $G_i = ShortWeakLiu(G_i)$     ▷ Adapt Embedding $\varphi$ according to edge replacements
    end if
end for
$G^* \leftarrow Merge((G_i)_{i\in[k]})$      ▷ Merge all EUGs as described in Theorem 2.0.3
$\varphi^* \leftarrow Merge((\varphi_i)_{i\in[k]})$      ▷ Merge the mappings into one map
$UC \leftarrow$ convert $G^*$ to circuit with X,Y and ULUT
$Prog \leftarrow$ Derive Programming bits from $\varphi^*$ for each gate in $UC$
return $(UC, Prog)$
```

In order to accomplish this, we first fix some parameter $2 \leq k' \leq d$ which determines the amount of EUG copies we want to utilize in our construction. Following this, we generate an auxiliary graph as in Algorithm 4.2.

The fixed construction is then invoked with the auxiliary graph of our circuit. Finally the resulting UC is modified to rewire the edges from and to auxiliary poles towards the poles, to which the inputs and outputs were originally meant to go to in the circuit. Using Algorithm 4.2, the dynamic construction is given by Algorithm 4.3.

Note, that the edges of the auxiliary graph retain the slot information of the original graph, which is a necessary input to the EUG construction in the third step as part of the edge information, which is also copied in the rewiring process before converting into a circuit, to retain correctness.

We now denote similar to [Wig21] the notion of a leaky EUG to argue the correctness of the construction.

Definition 4.4.1. *For $P^+, P^- \in N^n$ the class $\Gamma_{P^+,P^-}(n)$ denotes the set of all DAGs with n nodes v_i with indegree P^+_i and outdegree P^-_i specifically. A* ***leaky EUG*** *for the specific vector G' is a DAG which can edge embed any $G' \in \Gamma_{P^+,P^-}(n)$.*

Considering, that if all fanins are the same as well as all fanouts, we have the fixed construction again, but with fewer EUG mergings. Since we cannot infer fanout information about the embedded graph from the leaky EUG anymore, we have a regular EUG again.

Algorithm 4.2 *CreateAuxiliary*(G, k)

```
    Input:  Topologically sorted DAG $G = (V, E)$
            Degree $k$
   Output:  Auxiliary graph $G^*$
$G' = (V', E') = (\emptyset, \emptyset)$
for Node $v \in V$ do
    $IN_v = \{in_i | 1 \leq i \leq \lceil \frac{\delta^+(v)}{k} \rceil - 1\} \cup \{v'\}$
    $OUT_v = \{out_i | 1 \leq i \leq \lceil \frac{\delta^-(v)}{k} \rceil - 1\} \cup \{v'\}$
    $V' = V' \cup IN_v \cup OUT_v$
end for
for Edge $e = (u, v) \in E$ do
    $u' = Pop(OUT_u)$
    $v' = Pop(IN_v)$
    $E' = E' \cup (u', v')$                    ▷ Also copy slot numbers from $e$ onto $e'$
    if $\delta^-(u') \leq k$ then
        $Push(u', OUT_v)$
    end if
    if $\delta^+(v') \leq k$ then
        $Push(v', IN_v)$
    end if
end for
$G^* = TopoSort(G')$      ▷ Move all $IN_v$ before $v$ and $OUT_v$ after $v$ in the topological order
return $G^*$
```

Lemma 4.4.1. *If $P^+ = P^- = 1 * \rho$ then $\Gamma_{P^+,P^-}(n) = \Gamma_\rho(n)$ and a leaky EUG for that class also is a regular EUG.*

It should be noted, that this construction case is not really useful, as the amount of needed auxiliary poles, and thus larger EUG size generally performs worse than the regular fixed construction with more mergings.

We now prove informally

Theorem 4.4.2. *The Graph of the UC produced in Algorithm 4.3 is a leaky EUG for $\Gamma_{P^+,P^-}(n)$ with size bounded by:*

$$3(n + \Delta)\log_2(n + \Delta) + \mathcal{O}(n + \Delta)$$

such that $\Delta = \Sigma_{i=0}^{n} \max(\lceil \frac{P_i^+ - k}{k} \rceil, 0) + \max(\lceil \frac{P_i^- - k}{k} \rceil, 0)$

Note, that the fundamental complexity of the underlying construction is not changed, as the construction of the EUG itself is not changed, but instead is re-parametrized.

Algorithm 4.3 *CreateInPlaceUC*(C, k)

```
Input:  Circuit $C$                                  ▷ Circuit to be embedded
        $k \leq \max\{\Delta(v) | v \in Graph(C)\}$  ▷ Number of mergings. $Graph(C)$ is
                                                     the graph representation of the Circuit
Output: Universal Circuit & programming
$C' \leftarrow CreateAuxiliary(Graph(C), k)$
$(UC, Prog) \leftarrow CreateFixedUC(C', k)$
for $p = Pole(OUT_u) \in UC$ do
    for $e \in outgoingEdges(p)$ do
        Find $v$ s.t. $e == (p, Pole(v))$
        copy outgoing slot information from$(OUT_v, v)$ onto $e$
        Rewire $e = (p, Pole(v)) \leftarrow e = (Pole(u), Pole(v))$
    end for
end for
$InputEdgeForwarding(UC)$                            ▷ cf. [Wig21]
Remove all the poles of the auxiliary nodes from the UC
return $(UC, Prog)$
```

Proof. Given a circuit C with graph representation $G = (V, E)$ with $|V| = n$ the auxiliary graph construction as above is of the size $n + \Delta$ due to the auxiliary nodes. The topological sorting and reordering enforces, that all auxiliary nodes that enter inputs into a node v are ahead of it, outputs after it respectively. The reconnection of edges to the main poles allows for the higher input / output embedding, because every edge embedded to head to/from an auxiliary pole, can be immediately extended by the new extra edges from reconnecting to the main pole. For any two graphs $G, G' \in \Gamma_{P^+,P^-}(n)$, the Algorithm 4.2 yields a graph with the same set of nodes (including auxiliary poles), but different edges. We now consider that the graph of our UC, produced for both graphs from the auxiliary graph by our regular construction and the reconnecting of extra edges, yields the exact same Universal Circuit, as we know that the number of auxiliary nodes per pole, and thus the amount of redirected edges in both graphs is the same. If we now are given an Edge in G that was connected to an auxiliary pole in the edge embedding, we simply extend / rewrite the beginning and/or end part (an edge can go from the auxiliary pole of one pole to the auxiliary pole of another) to use those extra edges at the beginning and/or end. Note that the slot numbers, which are then copied from the original graph G onto the first edge of each embedding, are essentially represented by the order of inputs/ouputs, which is either virtual information of the graph, which is only visible in the final Circuit description, or constitutes an actual ordering of the ingoing/outgoing edges of the pole. This information however is meaningless, as the order can be arbitrarily changed by adjusting the ordering of the truth tables and because two outgoing edges of the same slot will be embedded as separate edges, thus making it impossible to tell which set of edges have the same slot. We therefore cannot distinguish the UC of G from the one of G', as it is the same. Thus we cannot infer more information about G or G' from the Universal

Circuit without the programming, except for the arity information and overall size, and the graphs are both correctly embedded. □

Note, that now k' can serve as a hyper parameter to govern how much information we want to leak about the function within use, i.e., how leaky the EUG truly is. If we choose k' to be the maximum of both in and out degree over all poles, there will be no need for the creation of auxiliary poles and no edges will be redistributed; all the poles have the same in- and outdegree. If we choose $k' = 2$ (the minimal practical choice for Boolean logic) the likelihood for redistribution from auxiliary poles to occur drastically increases, and thus the information leakage about the size of each LUT and possible function within the topological order of the circuit.

To also give an example of how such an attack might work, consider the scenario where $k' = 2$ and the first two poles of an exemplary circuit to be embedded into a UC have 2 and 3 outputs respectively. If it now occurs that the next pole in the topological order has exactly 5 inputs due to our dynamic construction, we know that the first two poles must feed all their outputs into this pole, as the only topological inputs before the third pole are these two, and it "consumes" all the outputs of the first two poles. If the UC was created chosen the fixed approach or $k' = 5$, then the same circuit could have been embedded, but as all poles hold 5 inputs and outputs, the attacker could not be certain whether the edges in question might be wired to some later part of the circuit all though this is more wasteful as some parts of the circuit do not truly contribute to the result of the computation but are instead dead leads running into nowhere by the design of the UC.

5 Evaluation

In order to analyze the performance of the proposed methods, we compile a set of selected circuits from cryptographic utility (block ciphers, hashes) to computational functions (addition, multiplication) as well as randomized circuits, and analyze their size in terms of the amount of AND gates, used X and Y gates, the distribution of Look Up Table sizes and compare the sizes to the implementation of [Wig21]. We introduce methods to build multi input-multi output gate circuits from multi input-single output circuits.

5.1 Construction of Multi Input-Multi Output Gate Circuits

In order to benchmark circuits with specific functionality, e.g., addition, we need to give a mapping from LUT circuits with singular outputs to a more condensed representation with multiple outputs. It is important, that the new representation of the circuit preserves/reduces the fanin/fanout over all gates in the circuit, as well as significantly reduce the amount of needed gates for the same functionality. This is both necessary to make an actual use of the multiple outputs (less gates require smaller EUGs in the UC construction and therefore yield smaller UCs overall) as well as to counteract the implicit negative effects of the multiple output UC construction in the fixed setting.

Consider a circuit in which all gates except one have 2 inputs, 1 output and 2 usages of this output. This one different gate might now have 3 inputs, 2 outputs and overall 4 usages of those 2 outputs. Just because of this single gate, the derived UC needs 3 EUG copies while most of the circuit only made use of 2. Furthermore every ULUT in the circuit will now be of AND gate size $(2^3 - 1) * 2 = 14$, as the fixed approach enforces all ULUTS to be of the same maximum size inside of the circuit. These compounding factors often lead to Universal Circuits which perform worse than constructions of multiple inputs and singular outputs [Wig21], simply by the fact that, to the authors, there is no known construction which yields smaller multi input-multi output circuits. Note that the dynamic construction does not suffer from these problems, as it has precisely this "shrink-wrapping" property, which allows for individual ULUT sizes as well as only needing two EUG copies, with the extra auxiliary gates adaptively wrapping the size of the EUGs around the interfaces of the gates by the auxiliary poles. Two different approaches have been considered to construct multiple input/output circuits. Note that this trade-off also precisely has the security implications that for full function hiding (PFE) the ULUTs should all have the same sizes whereas in SPFE this may be more leaky by having adaptive ULUT arities in the UC.

5.1.1 Multi Input Gate Collection [DKS+17]

The approach introduced by [DKS+17] merges multiple input gates, which lie on the same topological depth (i.e., one does not have a path to the other), if they have some overlapping use of input wires. A parameter is given which controls how many gates are merged at most. This approach does not allow for precise control of fanin/fanout and retains the same depth in the circuit.

5.1.2 Adaptive Gate Merging

With this approach we propose an improvement to merging gates into smaller multi input-multi output circuits.

Algorithm 5.1 $CreateMultiInputOutputCircuit(C, \max_{fanin}, \max_{fanout}, \max_{outputs})$

```
     Input:  multi input-single output circuit $C$
             maximum allowed fanin per gate
             maximum allowed fanout per gate
             maximum amount of different outputs per gate
    Output:  multi input-multi output circuit $C'$ so that $\forall x : C(x) == C'(c)$
 1: $C' = (g_i =$ i. gate in circuit$) \leftarrow topo - sort(C)$
 2: $modification \leftarrow True$
 3: $i \leftarrow 0$
 4: for $h = 0$ to $N$ do
 5:     modification $\leftarrow False$
 6:     for $j = i + 1$ to $|C'|$ do
 7:         if $CanMerge(g_i, g_j, \max_{fanin}, \max_{fanout}, \max_{outputs})$ then      ▷ cf. Algorithm 5.2
 8:             $C' \leftarrow C' - \{g_i, g_j\} \cup \{MergedGate(g_i, g_j)\}$            ▷ cf. Algorithm 5.3
 9:             $modification \leftarrow True$
10:         end if
11:         $i \leftarrow (i + 1) \bmod |C'|$
12:         if not  modification  and  $i = 0$ then
13:             Break
14:         end if
15:     end for
16: end for
17: return $C'$
```

The Algorithm 5.1 tries iteratively to merge lookup tables whose cumulative size follows given restrictions to inputs, outputs and fanout so long as their merging can be performed while respecting topological order. Algorithm 5.2 checks if they can be merged, which is either the case if they are of the same topological depth (parallel and thus not interfering with one another) or in immediate succession (there is a result shared between the two gates

Algorithm 5.2 *CanMerge*$(x, y, \max_{fanin}, \max_{fanout}, \max_{outputs})$

```
Input:  LUT Nodes $x$ and $y$
        maximum allowed fanin per gate
        maximum allowed fanout per gate
        maximum amount of different outputs per gate
Output: $True \iff x$ and $y$ may be merged within the given constraints
1: $a \leftarrow TopoDepth(x)$  ▷ Compute depth of the gate in the circuit according to topological order
2: $b \leftarrow TopoDepth(y)$
3: $in \leftarrow Inputs(\{x, y\})$  ▷ Set of inputs that are not an output of $x$ or $y$
4: $out \leftarrow Outputs(\{x, y\})$  ▷ Set of outputs of $x$ and $y$ that are used in gates other than $x$ or $y$
5: $uses_{out} = NumberOfUses(out)$  ▷ Number of total uses of all outputs outside of $x$ and $y$
6: if $|in| \geq \max_{fanin}$ or $|out| \geq \max_{outputs}$ or $uses_{out} \geq \max_{fanout}$ then
7:     return $False$
8: end if
9: return $a = b$ or $(|a - b| = 1$ and $in \cap out \neq \emptyset)$
```

Algorithm 5.3 *MergedGate*(x, y)

```
Input:  LUT Nodes $x$ and $y$
Output: Merged node of $x$ and $y$
1: $in \leftarrow Inputs(\{x, y\})$
2: $out \leftarrow Outputs(\{x, y\})$
3: for $i = 0$ to $2^{|in|}$ do
4:     for $j = 0$ to $|out|$ do
5:         $inputs_{bin} \leftarrow Bin(i)$
6:         $o_{i,j} \leftarrow j$.output of evaluation of subcircuit $C^* = \{x, y\}$ on $inputs_{bin}$
7:     end for
8: end for
9: $resultingnode \leftarrow newNode$
10: $resultingnode.inputs \leftarrow in$
11: $resultingnode.outputs \leftarrow out$
12: $resultingnode.programming \leftarrow \{o_{i,j} | i \leq 2^{|in|}, j \leq |out|\}$  ▷ these values determine the new LUT programming
13: return $resultingnode$
```

and no node that lies on a path between them). Note that the topological ordering is built up and preserved in order to allow for significantly faster computation of topological depth. If two gates may be merged, they are removed from the circuit, and replaced by a new larger gate, that encompasses the functionality of both old gates. Algorithm 5.3 creates the new larger LUT gate, by determining the border of wires which are used by the two merged nodes and are output, as well as determining the larger set of programming bits. Note that this merging may also reduce fanout of some of the outputs, as they may have had one of their uses in one of the two removed nodes.

5.1.3 Generating Random Circuits

It is also relevant to consider creating randomized circuits to get an overview of the amortized performance of the UC construction. This is done in a similar manner to [Wig21] while also introducing multiple outputs through a minor modification (outgoing edges randomly choose from a random set of outputs which have random programming bits each). We can procede along the following lines.

1. Given amount of total nodes at most N and fanin/fanout parameters ρ_+, ρ_- create a matrix of size $N \times N$.
2. Fill the matrix to be a random adjacency matrix by coin flipping every field in the upper right triangular without the diagonal (this ensures the resulting graph to be a DAG).
3. Adjust the matrix such that each column has at most ρ_- one's and each row at most ρ_+ one's.
4. Convert the given adjacency matrix into a graph.
5. Pick the largest connected sub graph of this graph and discard remaining disjoint segments.
6. Every node without ingoing edges is treated as an input to the circuit, every node without outgoing edges as an output.
7. Assign every node with ingoing and outgoing edges random programming bits (assign which outgoing edge receives which output at random) such that the total amount of outputs is random and the amount of inputs is exactly the amount of ingoing wires.
8. Write the resulting circuit to an arbitrary format (e.g. *SHDL*, *Bristol* etc).

5.2 Analysis of Circuit Sizes

We consider the circuits listed in Tab. 5.1, which where previously benchmarked by [Wig21] to evaluate the size performance of multi input-single output Universal Circuits as compared to the classic constructions of Universal Circuits. The circuits used by [Wig21] where sourced from [TS] [1] for cryptographic examples as well as [DKS+17] for computational examples. All the circuits of [TS] are formatted in the *Bristol* format and all circuits of [DKS+17] are given in the *VHDL-Verilog* language. The Verilog language is a platform agnostic programming

[1] https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html

language for describing digital hardware, which allows, by a plethora of synthesis and verification tools, a very intuitive and efficient description of arbitrary Boolean functions within submodules. This notion of submodules may be of great benefit, as this is precisely the beneficial encapsulation mechanism we want to exploit with multi input-multi output LUTs. If a synthesis tool would exist, that would directly map all the submodules of a given module into LUTs, this would yield a simple and very effective tool for creating multi input-multi output gate circuits for our compiler. An example of this can be seen in the synthesis of our Ripple Carry Adder Circuit, in which each Full Adder may be given as a submodule, such that all the submodules need only be connected by wiring. Unfortunately, such a tool is not known to us at the time of this writing. As described in § 5.1.3, the random/procedural circuits can be given in any format. We choose to create them in a supplementary tool and describe them in the *SHDL* format, allthough they may also be translated into the *BENCH* circuit benchmarking format [**benchform**; Qin06] or similar. We will directly compare the sizes of our circuits to those of the previous compiler to gauge the improvement provided by the addition of multiple outputs.

Circuit	Source	# Boolean Gates
AES-expanded	[TS]	27 692
MD5	[TS]	77 861
SHA-256	[TS]	236 111
SHA-1	[TS]	106 601
Ripple Carry Adder $128 \times 128 \to 64$	[DKS+17]	1 522
Karatsuba Multiplier $32 \times 32 \to 64$	This work	10 447

Table 5.1: Circuits compiled in the multi in-out setting

All of the external example circuits are described as regular Boolean circuits, consisting of Boolean gates such as XOR, AND, NOT, etc. If the circuits are not produced procedurally, they are synthesized into the *BLIF* [Ber] circuit model description format by the **Yosys** Suite [WGK13]. Then, a transformation is applied to merge multiple sets of gates into LUTs with multiple inputs and singular outputs. **Yosys-ABC** provides tools to perform this transformation according to several different heuristics.

We use the script of Fig. 5.2 as arguments to perform the transformation into multi input-single output circuits. Note that we fix the LUT input size to at most 8, as this is a tried and tested number which is plausible for execution and storage of circuits ([Wig21] showed that larger sizes are detrimental due to the exponential growth in programming bit sizes).

In an intermediary pre-processing pass for benchmarking, as lined out by Fig. 5.1, we need to reduce the fanout of the circuits still, as we otherwise would cause the creation of too many EUGs. This is exasperated by the fact, that merging multiple gates, effectively adds the amount of outgoing wires up. The fanout reduction is performed by postprocessing the multi input-multi output SHDL graph and adding in copy gates.

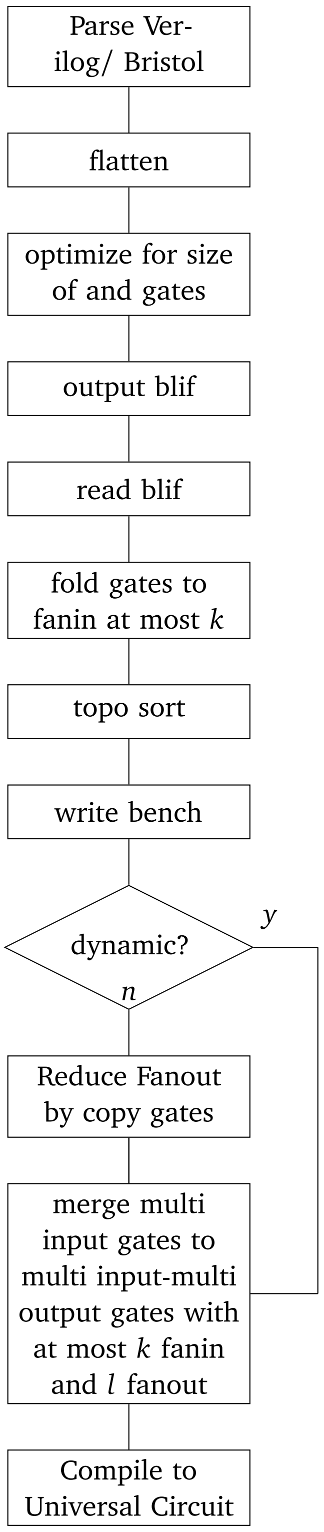
Parse Verilog/ Bristol
flatten
optimize for size of and gates
output blif
read blif
fold gates to fanin at most k
topo sort
write bench
dynamic?
y
n
Reduce Fanout by copy gates
merge multi input gates to multi input-multi output gates with at most k fanin and l fanout
Compile to Universal Circuit

Figure 5.1: Workflow for benchmarking

```
yosys:
read_<<verilog/bench>> <path to circuit file>
flatten ; opt; proc; opt; memory; opt; fsm; opt; techmap; opt;
write_blif <outputpath>

yosys-abc:
read_blif <outputpath>
strash; dc2; fraig; retime; strash;
dch -f; if -K 8 -F 200 -a; mfs;
topo
cec
print_stats
write_bench <outputpath>
```

Figure 5.2: Yosys scripts for multi input-single output LUT synthesis

The *BENCH* format file is now translated into *SHDL* in order to apply a custom transformation script that creates multiple outputs. This is done by collecting sets of LUTs that share some inputs and are of the same topological order (this is necessary to prevent the creation of cycles) and merging them into a single node. As the inputs of a gate might change, we need to reprogram the truth tables for each output, such that the inputs which are not important for one specific output, but are included for other outputs, are treated like don't care values, i.e., they have no influence on that particular output; the truth table is still described on all inputs, including those without influence, and the given output value is the same value we would choose if we evaluated the old gate of that output on the inputs with influence. Note that a parameter might be used to control the amount of outputs, a single LUT can have at most. We finally output the circuit in the *SHDL* format and compile it with the UC compiler both in the dynamic (in place) and fixed approach with either the Valiant [Val76] or Liu et al. [LYZ+21] EUG construction.

Another option in compiling a circuit, given as compiler flag *—s* is the *strong gate description hiding*. This flag disables the reuse of output values in the fixed construction, by copying the truth table of an outgoing wire as many times as the original output was used, thus hiding each of the uses as a different output of the LUT. This extra hiding, which makes the security more strict, comes at the cost of having potentially larger LUTs and thus increasing the size of our UC. The adjustment of the strong gate description hiding, along with the selection of used mergings in the dynamic approach, however, should only be done under careful consideration, as it has both security and performance implications, which might lead to undesirable leakage of properties of the function.

An overview of compiler options can be seen in Tab. 5.2.

For measuring the AND gate sizes, we tally up, all the gates of the circuit description file, where we have

Flag	Description
$-f$	Input circuit file in *SHDL* format
$-l$	Use Liu's [LYZ+21] EUG construction
$-i$	Use in-place (dynamic) construction
$-s$	Use strong gate description hiding
$--mergings <k>$	Control amount of EUGs used in dynamic approach (default 2)
$-n$	Disable randomized correctness check

Table 5.2: Flags for the UC compiler with description

1. *#AND(X-Gate)* $= 1$
2. *#AND(Y-Gate)* $= 1$
3. *#AND(m → n ULUT)* $= (2^m - 1) * n$

These sizes derive from our descriptions of ULUTs as well as [AGKS20].

5.3 Circuits for Arithmetic

We now investigate the results of the constructions empirically. The size results generally are the number of AND gates used in the UC unless described otherwise. Whenever we refer to just *Multi In* constructions in our data, we compiled the circuit in the framework [Wig21] for comparison. Also, whenever a black line is added to the plot, it describes the number of AND gates for the classical boolean approach without LUTs. Also note that whenever we refer to Input/Output arity x in any plot, we sample the smallest achieved UC size over all possible circuit compilations of the circuit with $(k, l) \in [8] \times [8]$ as in Fig. 5.1 with $\max(k, l) = x$.

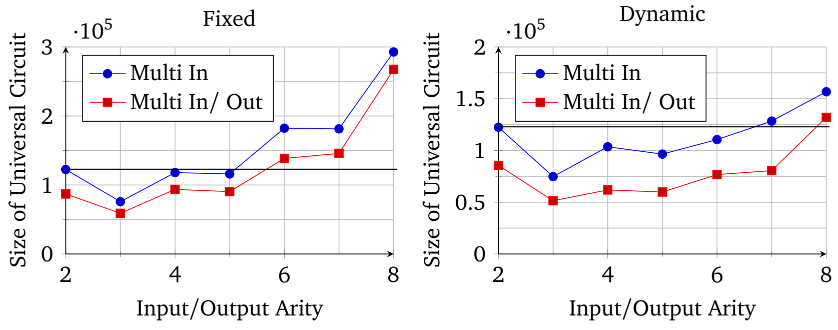

Figure 5.3: Smallest size of a UC of a 256 Bit RCA with ULUTs of at most x In/Outputs per ULUT with Valiant EUGs for the fixed construction

For the 256-RCA circuit we observe the sizes reported in Fig. 5.3. As expected, for ripple carry adders, the optimal size is achieved in the dynamic construction with fanin 3 and fanout 2 (set of Full Adders) with some gates of lower fanin and fanout. This is due to the fact that we can use 256 ULUTs, one per Full Adder. Each Full Adder ULUT requires $(2^3-1)*2=14$ AND gates, a similar Boolean representation requires 15 gates per full adder. Also, because the topology inside of the LUT is not hidden, we save on the size for EUGs compared to the classical boolean setting of [AGKS20]. For the fixed construction, the best size is achieved if all LUTs are of arity $3 \rightarrow 2$. The dynamic approach, which is SPFE is better than the PFE approach (fixed construction) by approximately 13%. This is the case, because for a more varied gate size distribution, we can push down the number of needed LUTs further, while using the natural benefit of the adaptive input/output sizes of our dynamic construction. There is an $\approx$ 30% size improvement for Ripple Carry Adders over the dynamic construction of [Wig21] in our dynamic construction with both using Liu's EUG. For the fixed construction we achieve a $\approx$ 18% improvement over [Wig21] in total size. Note that this improvement for the same fanin/fanout k gets smaller with larger k, as the circuit representation with $k \geq 3$ fanin/fanout gets less efficient. We would have to reduce the overall size of the circuit to be embedded by the factor k to yield a better improvement for larger input/output sizes, but as of the time of this writing, it is not possible.

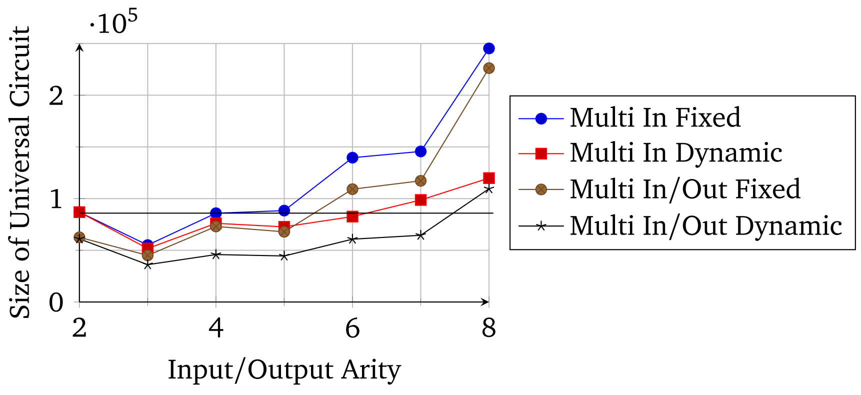

Figure 5.4: Size of a UC of a 256 Bit RCA with ULUTs of at most x In/Outputs per ULUT with Liu EUGs for the fixed and dynamic construction

As Fig. 5.4 lines out, using Liu's Construction instead of Valiant yields a general improvement in sizes independent of other parameters. The Valiant construction however is included as this allows us to compare our improvement towards [AGKS20], as the construction of [Wig21] with $k=2$ is the same construction. We therefore have the same improvement in relation to [AGKS20].

If we compare the distribution of the contribution of gate types to the size of a circuit in Fig. 5.5, the difference in the contribution of the ULUTs becomes noticeable. In the fixed construction, the ULUTs make up a larger amount of the gates, as many LUTs are padded out to the largest occurring ULUT. In the dynamic construction, a larger percentage of the size is given by X and Y gates. This is caused by the larger EUGs which are needed to accommodate auxiliary poles, while having fewer and individually sized LUTs, thus, changing the distribution.

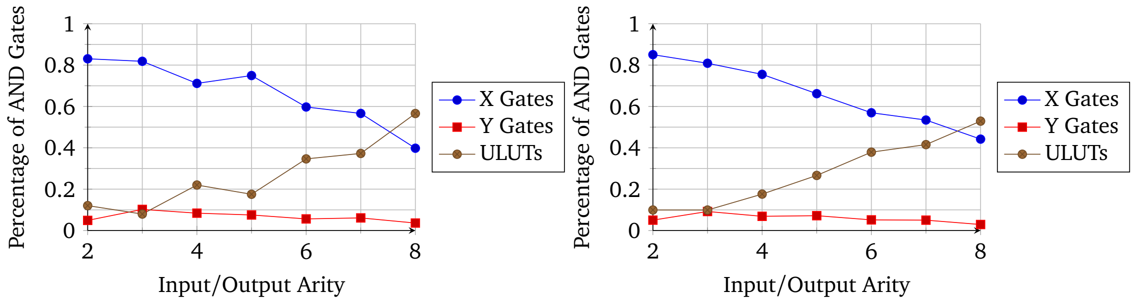

Figure 5.5: Percentage of AND gates by gate types in a UC of the 256-RCA

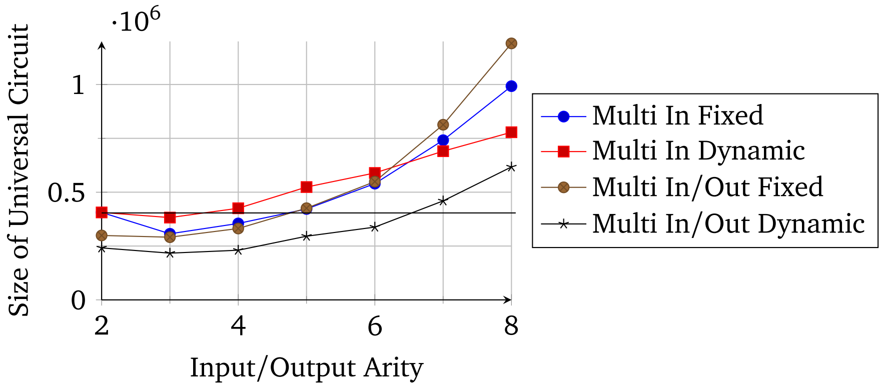

Figure 5.6: Size of a UC of a Karatsuba multiplier with ULUTs of at most x In/Outputs per ULUT with Liu EUGs for the fixed and dynamic construction

Another important circuit for arithmetic is the Karatsuba multiplier. For the Karatsuba multiplier, we observe in Fig. 5.6 a similar improvement of $\approx 45\%$ for the dynamic construction as well as 5% for the fixed approach. The dynamic approach is approximately 25% smaller than the fixed approach.

5.4 Cryptographic Circuits

We will benchmark cryptographic circuits only with Liu's EUG construction, as it has the same or smaller sizes than Valiant's construction for any circuit. The evaluation of cryptographic circuits yields the results in Fig. 5.7 for AES, Fig. 5.8 for SHA-256, Fig. 5.9 for SHA-1 and Fig. 5.11 for MD5.

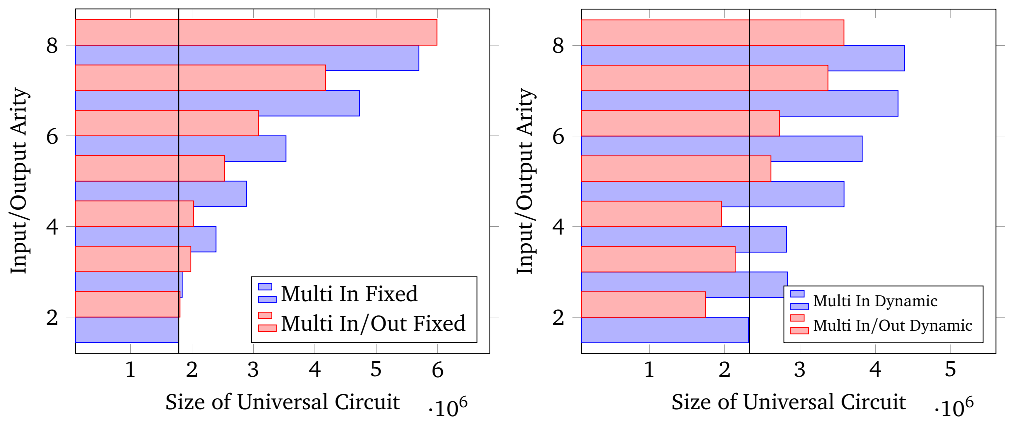

Figure 5.7: Size of a UC of the AES block cipher with ULUTs of at most x In/Outputs per ULUT with Liu EUGs for the fixed and dynamic construction

For the AES block cipher, the result shows no improvement in size for the fixed construction. The dynamic construction achieves an improvement of $\approx 25\%$ in the case of $k = 2$. The overall optimal construction for AES is now the $2 \to 2$ ULUT dynamic construction with Liu EUGs.

For the fixed construction in Fig. 5.8, SHA-256 does not achieve a strong improvement for the same reasons as MD5 (the improvement is around 5%). The dynamic construction meanwhile improves by 21% in the case of at most $4 \to 4$ LUTs.

The size improvement is observably lower under cryptographic circuits than other types of circuits. This is largely due to the strong interconnectedness of gates inside of the circuit, which prevents a much smaller description in terms of LUTs with multiple outputs, because merging many LUTs with large fanout themselves, creates LUTs with even larger fanouts.

As Fig. 5.10 demonstrates, the gate sizes are not evenly distributed in the case of AES, which leads to a weaker improvement. Because many gates are single input-single output, but with

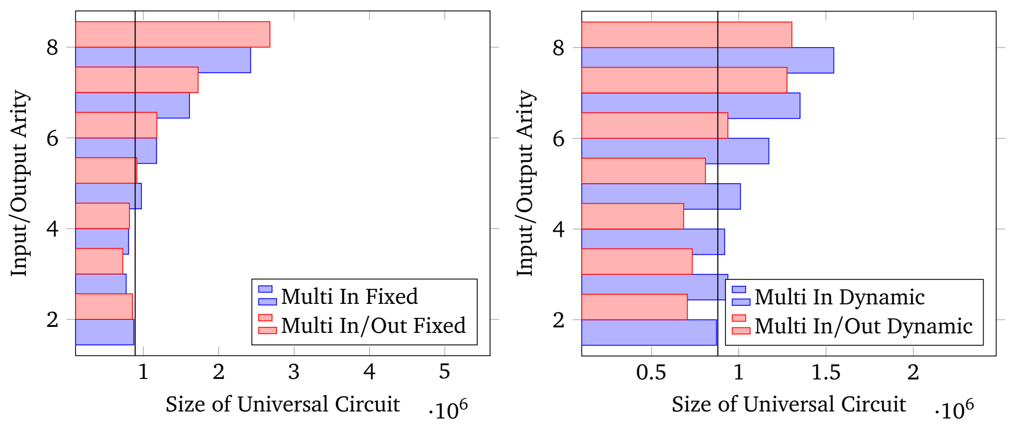

Figure 5.8: Size of a UC of the SHA-256 Hash with ULUTs of at most x In/Outputs per ULUT with Liu EUGs for the fixed and dynamic construction

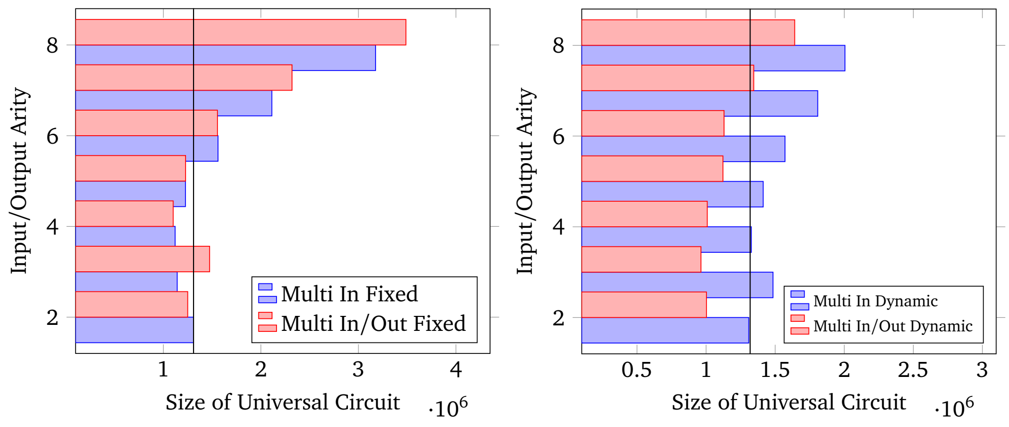

Figure 5.9: Size of a UC of the SHA-1 Hash with ULUTs of at most x In/Outputs per ULUT with Liu EUGs for the fixed and dynamic construction

many wire usages, the current tooling struggles with integrating them into larger ULUTs as to distribute the size more evenly. Ideally for the fixed construction, we want to achieve a unimodal distribution in the lower minor (small input and small output sizes), which may tamper out to larger fanin/fanout to only a small degree, to avoid over-allocating large ULUTs for a circuit which might have a simpler description. For the dynamic construction however, a more varied distribution is more beneficial, as we thus make use of the adaptive fanin/fanout nature of the approach (otherwise we would have to increase the number of mergings, effectively transforming the approach gradually into the fixed approach). By Handcrafting the AES circuit from $2 \rightarrow 1$ ULUTs for the XOR operations and $8 \rightarrow 8$ ULUTs for the S-Boxes, we may achieve better sizes, but this could no longer be tested due to a lack of time and because the used tools could not accomplish such a construction on their own. Under all cryptographic circuits SHA-1 achieves the largest improvement over all at $\approx 26\%$ in the dynamic construction. For cryptographic circuits it is also not always the case, that the

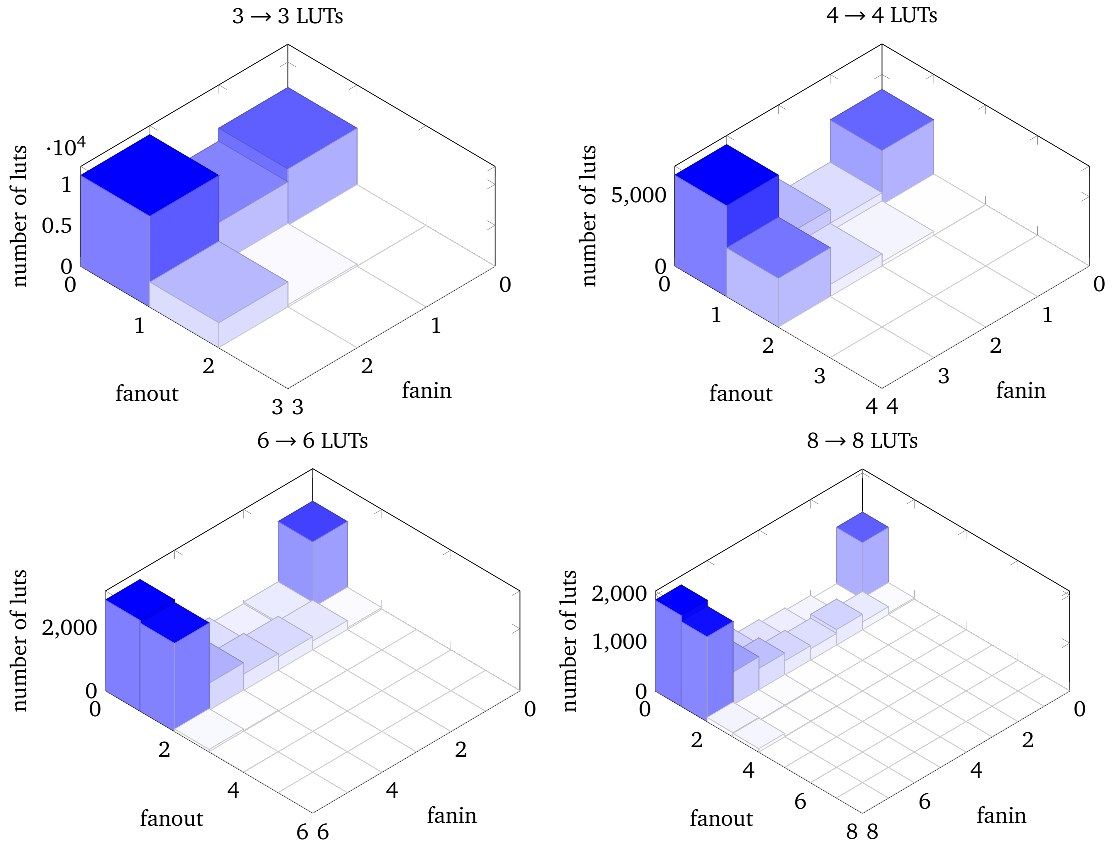

Figure 5.10: Distribution of LUT sizes for different maximum fanin/fanout LUT gates for the AES block cipher circuit.

dynamic construction outperforms the fixed construction. This is the case if the distribution of gate sizes is not mixed, thus needing a larger depth in case of the dynamic construction.

For the fixed construction of MD5, only if we choose $k = 4, 5, 6, 7$ we get a small improvement over the classic construction, but no overall improvement, as the $3 \rightarrow 1$ construction still yields the smallest size. Note that we can still perform this construction in the new compiler, thus producing at least the same size, as multi input-single output LUTs are a special case of our constructions. The worse performance in MD5 for the fixed construction can be explained by both a combination of the test methodology and the inherent problem of the method. As we have merged multiple gates with large fanout, we get a single gate with even larger fanout but of different output wires. This results in the necessary creation of many copy gates per output (currently we do not yet create copy gates that can forward multiple different results). This drastically increases the amount of gates to be embedded and thus the possible improvement being negated by the much larger size of the EUGs. A solution would either be to support copy gates of that kind or to provide circuits of better quality or even further rework the testing environment to prevent the merging of to many gates and to tactically

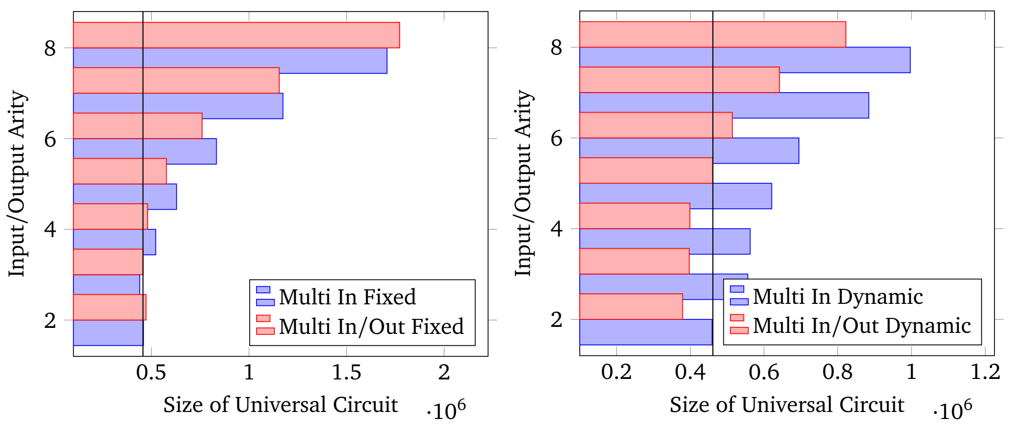

Figure 5.11: Size of a UC of the MD5 hash with ULUTs of at most x In/Outputs per ULUT with Liu EUGs for the fixed and dynamic construction

better distribute the fanout (this is precisely why submodules in the RCA contribute to the better performance). As can be seen in Fig. 5.11, the dynamic construction does not suffer from this problem, as it does not have to provide copy gates. For the dynamic construction, we achieve an overall improvement of $\approx 17,5\%$ in size.

5.5 General Circuits

We now want to benchmark circuits, which are not directly linked to a real world application, but are instead procedurally generated. These more general circuits are either random or constant, meaning that we either have up to n gates with varying numbers of inputs, outputs and uses, or exactly n gates which all have the same input and the same output number (and same number of uses for each output). For these more general circuits we can describe in a clearer fashion the nature of our construction methods and highlight their performance under more ideally controlled conditions.

If we compile randomized circuits with at most 10000 gates, 1000 inputs and outputs each, and fanin/fanout parameters as given in Fig. 5.12, the dynamic construction outperforms the fixed construction in all cases, except the case $2 \to 2$ where the performance stays the same. The largest improvement is observed in the $8 \to 8$ case, where an improvement of factor 20 is observed (a size improvement of $\approx 95\%$). An important observation is also, that the dynamic construction grows very slow compared to the fixed construction with increasing fanin, almost staying constant in case of low fanouts. The fixed construction however shows accelerated growth in size, starting from fanout 4.

Compiling under the same parameters, but with constant gate sizes (all gates in a circuit have the same fanin/fanout), the opposite observation can be made in Fig. 5.13. In all cases, the size of the circuits grow accelerated with growing fanin. This acceleration in growth

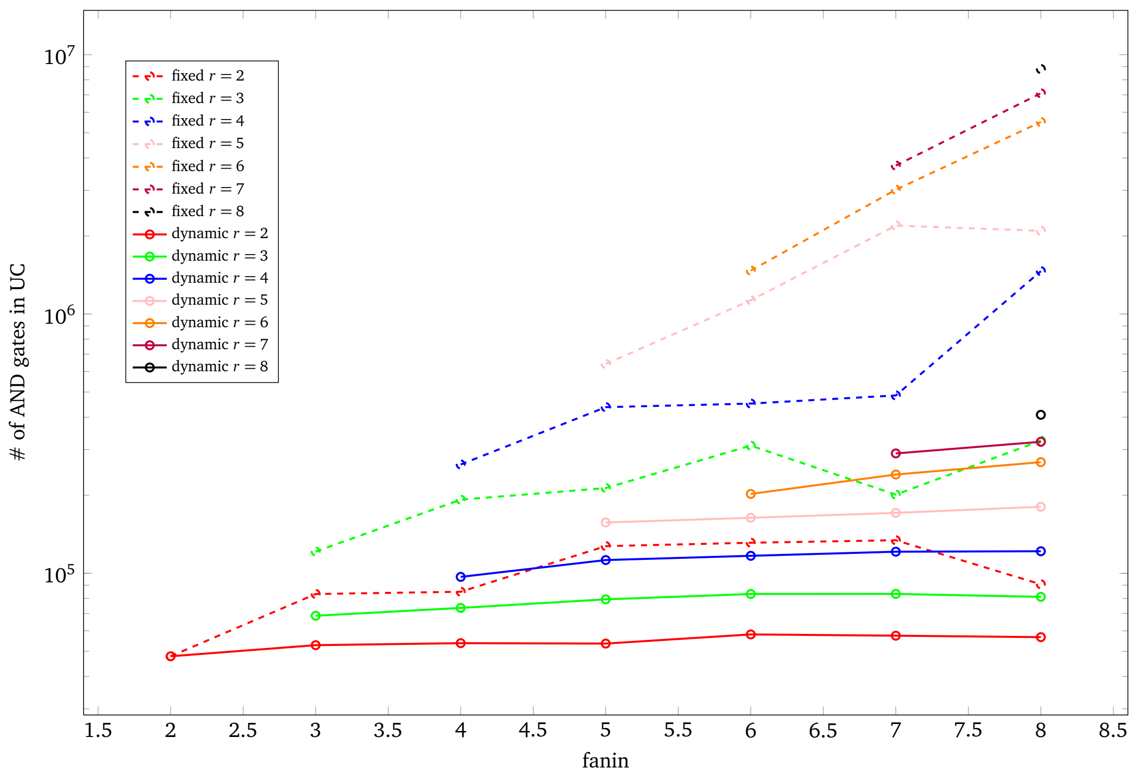

Figure 5.12: Size of UCs (#AND gates) of a random circuit with atmost 1000 inputs, 10000 LUTs with arity atmost $x \to r$ and 1000 outputs (log scaled)

is the same in both constructions. The overall improvement for the dynamic to the fixed construction varies by $\approx 7-46\%$, except for the case $2 \to 2$ where both constructions perform the same again.

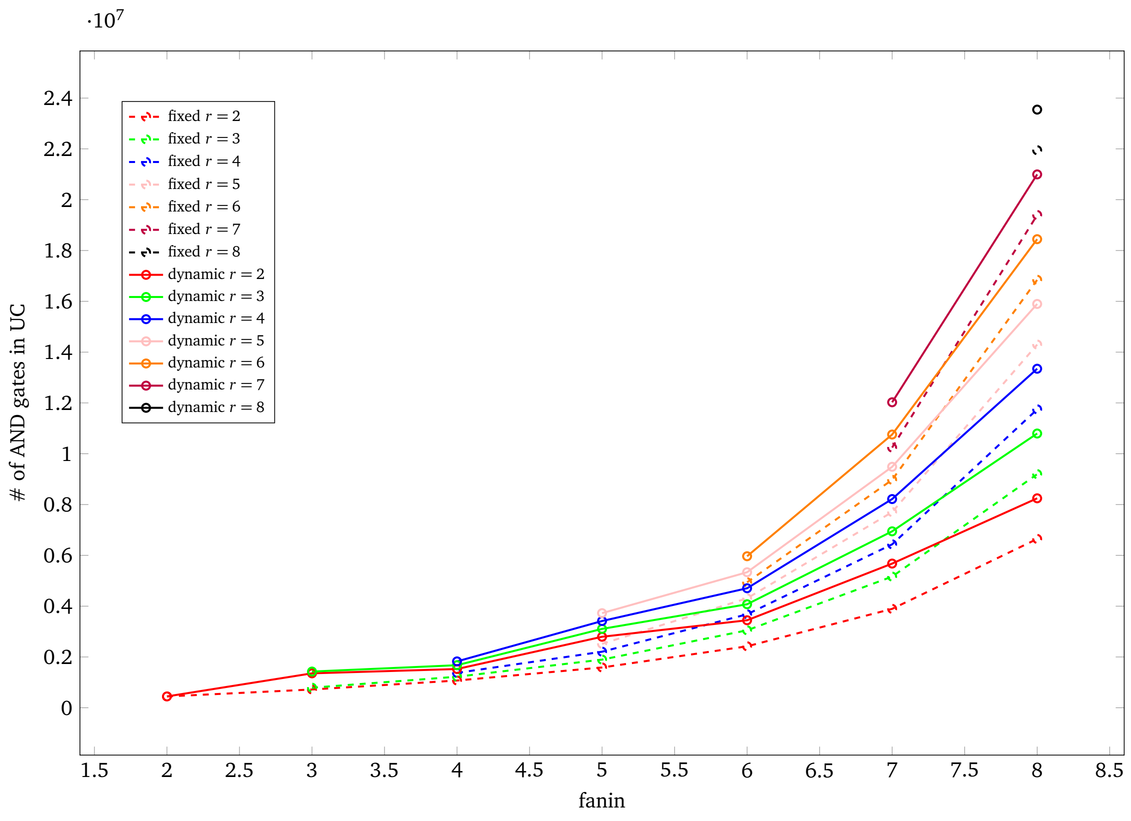

Figure 5.13: Size of UCs (#AND gates) of a constant circuit with 1000 inputs, 10000 $x \rightarrow r$ LUTs and 1 output

6 Conclusion

In this work we have given two methods for constructing multi input-multi output gate Universal Circuits of a smaller size than the state-of-the-art on the basis of leaking arity information of submodules. We achieved an improvement in the overall size of the UC for all circuits of arithmetic and cryptographic circuits tested for our dynamic construction by $\approx 33\%$. With our fixed construction, most of the circuits are improved by about $\approx 15\%$ in size, while not all circuits have an improvement, due to their topology and our current testing methodology. We argued the general use of these constructions both in terms of their complexity as well as in measurements of more generalized performance on procedural circuits. We provided a method for creating multi input-multi output gate circuits from Boolean circuits, to allow easy use of our framework, especially for circuits, which are not yet given in an optimized submodule description.

6.1 Implications and extra Observations

As noted prior, the provided framework allows to define a gradient of secrecy where we may control if we want to leak some information about our circuit or not and to what extent. It is however, difficult to pin down precisely how much information we end up leaking, i.e., how much the search space of all possible functions is reduced by leaking arity information of gates and reusage of partial results. The practical benefit comes from having to evaluate less outputs in a LUT and smaller EUGs, thus decreasing the circuit size and accelerating the evaluation of the secure multiparty computation protocol. The provided framework shows heavy dependance on the underlying topology of the circuit to be embedded, as well as the actual ability to significantly collect gates in a circuit to multiple inputs and outputs, for its improvements. Therefore, few uses of partial results inside of a circuit, as well as a propper boundary of abstraction into submodules (which would then ideally be small LUTs in a later synthesis) should be considered, when designing larger circuits to be embedded with these methods.

6.2 Future Work

As already noted earlier, one helpful tooling for achieving more optimal UCs, could be a Verilog technology mapping, which flattens submodules only up to the first level of hierarchy, and collects all the remaining submodules into LUTs, finally immediately mapping the resulting circuit description into a LUT based circuit description. Current toolings, mostly dedicated to the FPGA market, primarily target multiple inputs, which hinders the ability to make use of multiple outputs due to a lack of circuits.

Another possible improvement lies in the dynamic construction method. Several auxiliary nodes yield a surplus of inputs/outputs, which essentially go to waste, because they no longer serve any function inside of the embedding, remaining only as artifacts to the security. A method for finding subsequent nodes in the embedding, which, under respect of topological order, may make use of these inputs/outputs. Thus we may create less auxiliary nodes, further improving upon the size, as this method optimizes the use of auxiliary poles as a more globalized resource.

List of Figures

List of Tables

Bibliography

[AGKS20] M. Y. Alhassan, D. Günther, Á. Kiss, T. Schneider. **“Efficient and Scalable Universal Circuits”**. In: *Journal of Cryptology* 33.3 (2020), pp. 1216–1271.

[Alo03] N. Alon. **“A Simple Algorithm for Edge-Coloring Bipartite Multigraphs”**. In: *Information Processing Letters* 85.6 (2003), pp. 301–302.

[Ber] U. Berkley. **“Berkeley Logic Interchange Format (BLIF)”**.

[BHR12] M. Bellare, V. T. Hoang, P. Rogaway. **“Foundations of Garbled Circuits”**. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security - CCS ’12*. ACM Press, 2012, p. 784.

[Die17] R. Diestel. **“Graph Theory”**. Vol. 173. Graduate Texts in Mathematics. Springer Berlin Heidelberg, 2017.

[DKS+17] G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, M. Zohner. **“Pushing the Communication Barrier in Secure Computation Using Lookup Tables”**. In: *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.

[DSZ15] D. Demmler, T. Schneider, M. Zohner. **“ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation”**. In: *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

[FAZ05] K. Frikken, M. Atallah, C. Zhang. **“Privacy-Preserving Credit Checking”**. In: *Proceedings of the 6th ACM Conference on Electronic Commerce - EC ’05*. ACM Press, 2005, pp. 147–154.

[FKSW19] S. Felsen, Á. Kiss, T. Schneider, C. Weinert. **“Secure and Private Function Evaluation with Intel SGX”**. In: *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop - CCSW’19*. ACM Press, 2019, pp. 165–181.

[FVK+15] B. A. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, S. M. Bellovin. **“Malicious-Client Security in Blind Seer: A Scalable Private DBMS”**. In: *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 395–410.

[GGPR13] R. Gennaro, C. Gentry, B. Parno, M. Raykova. “Quadratic Span Programs and Succinct NIZKs without PCPs”. In: *Advances in Cryptology – EUROCRYPT 2013*. Vol. 7881. Springer Berlin Heidelberg, 2013, pp. 626–645.

[GKS17] D. GÜNTHER, Á. KISS, T. SCHNEIDER. "More Efficient Universal Circuit Constructions". In: *Advances in Cryptology – ASIACRYPT 2017*. Vol. 10625. Springer International Publishing, 2017, pp. 443–470.

[GKSS19] D. GÜNTHER, Á. KISS, L. SCHEIDEL, T. SCHNEIDER. **"Poster: Framework for Semi-Private Function Evaluation with Application to Secure Insurance Rate Calculation"**. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 2541–2543.

[GMW87] O. GOLDREICH, S. MICALI, A. WIGDERSON. **"How to Play ANY Mental Game"**. In: *Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing - STOC '87*. ACM Press, 1987, pp. 218–229.

[GVW15] S. GORBUNOV, V. VAIKUNTANATHAN, H. WEE. **"Attribute-Based Encryption for Circuits"**. In: *Journal of the ACM* 62.6 (2015), pp. 1–33.

[HKRS20] M. HOLZ, Á. KISS, D. RATHEE, T. SCHNEIDER. "Linear-Complexity Private Function Evaluation Is Practical". In: *Computer Security – ESORICS 2020*. Vol. 12309. Springer International Publishing, 2020, pp. 401–420.

[KM11] J. KATZ, L. MALKA. "Constant-Round Private Function Evaluation with Linear Complexity". In: *Advances in Cryptology – ASIACRYPT 2011*. Vol. 7073. Springer Berlin Heidelberg, 2011, pp. 556–571.

[KS08] V. KOLESNIKOV, T. SCHNEIDER. "A Practical Universal Circuit Construction and Secure Evaluation of Private Functions". In: *Financial Cryptography and Data Security*. Vol. 5143. Springer Berlin Heidelberg, 2008, pp. 83–97.

[KS16] Á. KISS, T. SCHNEIDER. "Valiant's Universal Circuit Is Practical". In: *Advances in Cryptology – EUROCRYPT 2016*. Vol. 9665. Springer Berlin Heidelberg, 2016, pp. 699–728.

[LYZ$^+$21] H. LIU, Y. YU, S. ZHAO, J. ZHANG, W. LIU, Z. HU. "Pushing the Limits of Valiant's Universal Circuits: Simpler, Tighter and More Compact". In: *Advances in Cryptology – CRYPTO 2021*. Vol. 12826. Springer International Publishing, 2021, pp. 365–394.

[MS13] P. MOHASSEL, S. SADEGHIAN. "How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation". In: *Advances in Cryptology – EUROCRYPT 2013*. Vol. 7881. Springer Berlin Heidelberg, 2013, pp. 557–574.

[MSS14] P. MOHASSEL, S. SADEGHIAN, N. P. SMART. "Actively Secure Private Function Evaluation". In: *Advances in Cryptology – ASIACRYPT 2014*. Vol. 8874. Springer Berlin Heidelberg, 2014, pp. 486–505.

[PSS09] A. PAUS, A.-R. SADEGHI, T. SCHNEIDER. "Practical Secure Evaluation of Semi-private Functions". In: *Applied Cryptography and Network Security*. Vol. 5536. Springer Berlin Heidelberg, 2009, pp. 89–106.

[Qin06] J. QIN. **"LOGIC SIMULATOR FOR BENCH FORMAT"**. 2006.

[Sha49] C. E. SHANNON. **"The Synthesis of Two-Terminal Switching Circuits"**. In: *Bell System Technical Journal* 28.1 (1949), pp. 59–98.

[SS09] A.-R. SADEGHI, T. SCHNEIDER. "Generalized Universal Circuits for Secure Evaluation of Private Functions with Application to Data Classification". In: *Information Security and Cryptology – ICISC 2008*. Vol. 5461. Springer Berlin Heidelberg, 2009, pp. 336–353.

[TS] S. TILLICH, N. SMART. **"(Bristol Format) Circuits of Basic Functions Suitable For MPC and FHE"**.

[Val76] L. G. VALIANT. **"Universal Circuits (Preliminary Report)"**. In: *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing - STOC '76*. ACM Press, 1976, pp. 196–203.

[WGK13] C. WOLF, J. GLASER, J. KEPLER. **"Yosys-A Free Verilog Synthesis Suite"**. In: 2013.

[Wig21] A. WIGANDT. **"Private Function Evaluation for Multi-Input Gates"**. 2021.

[Yao86] A. C.-C. YAO. **"How to Generate and Exchange Secrets"**. In: *27th Annual Symposium on Foundations of Computer Science (Sfcs 1986)*. IEEE, 1986, pp. 162–167.

[ZYZL19] S. ZHAO, Y. YU, J. ZHANG, H. LIU. "Valiant's Universal Circuits Revisited: An Overall Improvement and a Lower Bound". In: *Advances in Cryptology – ASIACRYPT 2019*. Vol. 11921. Springer International Publishing, 2019, pp. 401–425.