Overlaying Conditional Circuit Clauses for Secure Computation

W. Sean Kennedy, Vladimir Kolesnikov, and Gordon Wilfong

Bell Labs, Murray Hill, NJ, USA
{william.kennedy, vladimir.kolesnikov,
gordon.wilfong}@nokia-bell-labs.com

Abstract. We improve secure function evaluation (SFE) by optimizing circuit representation of the function and designing new SFE protocols.

1) We propose a heuristic for constructing a circuit C_0, universal for a given set of Boolean circuits $\mathcal{S} = \{C_1, ..., C_k\}$. Namely, given each C_i, we view it as a directed acyclic graph (DAG) D_i by ignoring the Boolean gate functions of C_i. We embed $D_1, ..., D_k$ in a new DAG D_0, such that each C_i can be obtained by a corresponding programming of D_0 (i.e. by assignment of Boolean gates to the nodes of D_0). DAG D_0, viewed as a Boolean circuit with unprogrammed gates, is the $\mathcal{S}$-universal circuit C_0.

2) Our heuristic often produces C_0 significantly smaller than Valiant's universal circuit or a circuit incorporating all $C_1, ..., C_k$. Exploiting this, we construct new Garbled Circuit (GC) and GMW-based SFE protocols, which are particularly efficient for circuits with `if/switch` clauses.

Our GMW protocol evaluates 8-input Boolean gates at the same cost as the usual 2-input gates. This advances general GMW-based SFE, and is particularly useful for circuits with `if/switch` conditional clauses.

Experimentally, for a switch containing 32 simple circuits, our construction resulted in $\approx 6.1\times$ smaller circuit C_0. This directly translates into $\approx 6.1\times$ improvement in GMW SFE computing this switch. Recent state-of-the-art generic circuit optimizations from hardware design adapted to SFE report $10 - 20\%$ circuit (garble table) reduction.

Our SFE is in the semi-honest model, and is compatible with Free-XOR. We further show that optimal embedding is NP-hard.

Keywords: set-universal circuit, secure computation, garbled circuit, GMW

1 Introduction

Eliminating costs imposed by the circuit representation of Garbled Circuits (GC) and Goldreich, Micali and Wigderson (GMW) techniques has been an important open problem since the introduction of GC/GMW, with moderate success to date. There are two natural redundancies: GC/GMW must unroll the loops and evaluate *all* conditional (`if/switch`) clauses so as to hide which was

evaluated. (There is a third redundancy, protecting memory access patterns at the expense of processing entire input/array/data structure, which is applicable to both circuit and random-access representation. It is addressed by the influential work on Oblivious RAM (ORAM), started by [13] with then-"impractical" $\log^4 n$ factor overhead.)

Our work aims to solve the second kind of circuit redundancy. Constructing a small circuit C_0, universal for k given circuits, will allow to garble, transfer and evaluate just C_0, when computing `switch` on the k circuits. We believe this could be a useful tool in Secure Function Evaluation (SFE) compiler design.

On the cost of SFE and OT rounds. Our GC protocol *may* add a round of communication for each `switch` statement. We argue that the associated latency cost is negligible in many practical scenarios. This is because often the latency-related idling will be productively used for computation and communication in the same or another SFE instance. This is the case, e.g., in larger-scale SFE deployments, where many instances will be run in parallel, and where SFE *throughput* is a far more important parameter than *latency*. Note, in our GMW protocols there is no increase in the number of rounds due to `switch`.

1.1 Motivating applications

Functions with `switch` statements. In Blind Seer [25, 7], a GC-based private database (DB) system, private DB search is achieved by two players jointly securely evaluating the query match function on the search tree of the data. Blind Seer does not fully protect query privacy: it leaks the query circuit topology as the full universal circuit is not practical, as admitted by the authors. Applying our solution to that work would hide this important information, cheaply. Indeed, say, by policy the DB client is allowed to execute one of several (say, 2-50) types of queries. The privately executed SQL query can then be a `switch` of the number of clauses, each corresponding to an allowed query type. In Blind Seer the clause is selected by client's input, omitting some of the machinery. As a result, our Blind Seer application is particularly effective bringing improvements with as little as two clauses. Most of the cost of this DB system is in running SFE of the query match function at a large scale, so improvement to the query circuit will directly translate to overall improvement. We note that the core of the Blind Seer system is in the semi-honest model, but a malicious client is considered in [7].

(Part of) our work can be viewed as a heuristic aiming to construct a circuit universal for a set of functions $\mathcal{S} = \{C_1, ..., C_k\}$ ($\mathcal{S}$-universal circuit) at the cost less than that of full universal circuit. Thus (cf. next motivating example), our work may improve applications where we want to evaluate and hide which function/query was chosen by a player (say, which one of several functions allowed by policy or known because of auxiliary information).

SFE of semi-private functions (SPF-SFE) (see additional discussion in Sect 1.3) is a notion introduced in [26], bridging the gap between expensive private function SFE (PF-SFE) based on Universal Circuit [30, 19, 15, 22], and regular SFE (via GC) that does not hide the evaluated function. SPF-SFE

partially hides the evaluated function; namely, given a set of functions, the evaluator will not learn which specific function was evaluated. Indeed, often only specific subroutines are sensitive, and it is they that might be sufficiently protected by $\mathcal{S}$-universal circuit for an appropriate set of circuits $\mathcal{S}$. [26] presents a convincing example of privacy-preserving credit checking, where the check function itself needs to be protected, and shows that using $\mathcal{S}$-universal circuits as building blocks is an effective way of approaching this. Further, [26] builds a compiler which assembles GC from the $\mathcal{S}$-universal building blocks (which they call PPB, Privately Programmable Blocks). While [26] provides only a few very simple hand-designed blocks (see our discussion in Sect. 1.3), our work can be viewed as an efficient general way of constructing such blocks. We stress that in the SPF-SFE application, GC generator knows the computed function (clause selection), and our contructions are particularly efficient.

CPU/ALU emulation. Extending the idea of SPF-SFE, one can imagine a general approach where the players privately emulate the CPU evaluating a sequence of complex instructions from a fixed instruction set (instruction choice implemented as a GC `switch`). Additionally, if desired, instructions' inputs can be protected by employing the selection blocks of [19]. Such an approach can be built within a suitable framework (e.g., that of [26]) from $\mathcal{S}$-universal circuits provided by this work. We note that circuit design and optimization is tedious, and not likely to be performed by hand except for very simple instances, such as those considered in [26]. Instead, a tool, such as the one we are proposing, will be required.

In a recent work [32], a secure and practically efficient MIPS ALU is proposed, where the ALU is implemented as a `switch` over 37 currently supported ALU instructions evaluated on ORAM-stored data. TinyGarble [29] also design and realize a garbled processor, using the MIPS I instruction set, for private function evaluation. Our constructions would work with [32] in a drop-in replacement manner.

1.2 Technical Contribution

We improve secure function evaluation (SFE) by optimizing circuit representation of the function and designing new SFE protocols.

Our contribution consists of several complementary technical advances. Our most technically involved contribution is a novel algorithm to embed any k circuits in a new circuit/graph C_0. The embedding is such that a GC/GMW evaluation of any of $C_1, ..., C_k$ could be implemented by a corresponding evaluation of C_0. The size of C_0 could be smaller (6.1× smaller in our experiment) than the sum of the sizes of $C_1, ..., C_k$.

In the SPF-SFE case, when the evaluated clause C_i is circuit generator's private input, generator G simply sends the garbling implementing C_i.

For the general GC case, where clause is selected by an internal variable, we construct a new GC protocol with total communication cost $3kn_0+22n_0s$, where s is the computational security parameter and $n_0 = |C_0|$. For efficient embeddings, this compares favorably to state-of-the-art GC. The half-gates GC [34] of

the above k clauses will cost $2ns$, where $n = \sum_j |C_j|$. We show how GC branches can be nested, and we can apply our construction on each nesting level.

GMW SFE is more interesting. We make a novel observation that the cost of evaluation of the GMW gates is similar to that for a moderate number of boolean inputs, as that of a two-input gate. We exploit this to obtain an efficient GMW protocol for circuits with clauses whose cost per gate is similar to that of standard GMW.

Our approach is heuristic. We show that solving the graph embedding problem exactly is NP-hard.

Experimental validation and performance. For our experiment we considered 32 simple circuits implementing variants of several basic functions and generated an embedding 6.1× smaller than the standard circuit implementing the clauses. SPF-SFE and GMW of the corresponding `switch` is also improved by 6.1×.

For the GC case when clause is selected by internal variable, our 6.1× smaller embedding results in communication cost similar to that of classical Yao [33, 20], due to per-gate overheads. We expect our protocol to overtake optimal GC [34] for embeddings of slightly greater number of circuits or with further heuristic improvements.

1.3 Background and Related Work

Garbled Circuit, GMW, OT and Universal Circuit. Significant part of SFE research focuses on minimizing the size of the basic GC of Yao [33, 20], such as garbled row reduction techniques Free-XOR [18] and its enhancements FleXOR [17] and half-gates [34]. In contrast, in this work, we effectively eliminate the need for evaluation of entire subcircuits.

The GMW protocol [11, 12] had received less attention in the 2-party SFE literature than GC. In GMW, the two parties interact to compute the circuit gate-by-gate as follows. Players start with 2-out-of-2 additively secret-shared input wire values of the gate and obtain corresponding secret shares of the output wire of the evaluated gate. Addition (XOR) gates are done locally, simply by adding the shares. Multiplication (AND) gates are done using 1-out of-4 OT. For binary circuits, there are four possible combinations of each of the player's shares. Thus an OT is executed, where one player (OT receiver) selects one of the four combinations, and the other player (OT sender) provides/OT-sends the corresponding secret shares of the output wire. In the semi-honest model, the secret shares can be as short as a single bit. As in the GC approach, our work greatly reduces the size of the evaluated circuit.

Asymptotically, the best way to embed a large number of sub-circuit graphs into one circuit graph would be using the universal circuits [30, 19]. Respectively, for sub-circuits of size n, the size of the universal circuit generated by [30, 19] is $\approx 19n \log n$, and $\approx 1.5n \log^2 n + 2.5n \log n$. Very recent works [22, 15] polish and implement Valiant's construction. They report a more precise estimate of the cost (in universal gates) of Valiant's UC of between $\approx 5n \log n$ and $10n \log n$. Programming of universal gates may each cost 3 AND and 6 XOR gates (but

will not be needed in PFE and applications we discuss in this work). In sum, universal circuit approach becomes competitive for a number of clauses far larger than a typical `switch`. In a universal circuit embedding [30, 19, 22, 15], gates are embedded in gates and wires are embedded in pairwise disjoint chains of wires (with possible intermediate gates). Our embedding is more general allowing the chains of wires to overlap in a controlled way, leading to smaller container circuits.

Another technique for Private Function Evaluation (PFE) was proposed by Mohassel and Sadeghian [23]. They propose an alternative (to the universal circuit) framework of SFE of a function whose definition is private to one of the players. Their approach is to map each gate outputs to next gate outputs by considering a mapping from all circuit inputs to all outputs, and evaluate it obliviously. For GC, they achieve a factor 2 improvement as compared to Valiant [30] and a factor $3-6$ improvement as compared to Kolesnikov and Schneider [19]. Similarly to [30, 19], [23] will not be cost-effective for a small number of clauses.

Thus, (part of) our work can be viewed as heuristically constructing a circuit universal for a set of functions $\mathcal{S} = \{C_1, ..., C_k\}$ at the cost less than that of full universal circuit.

One of our contributions is an improved 1-out of-k OT algorithm to deliver the garbled `switch` clause to the evaluator. This is a special case of PIR (private information retrieval). We note existing sublinear in k work on computational PIR (CPIR) of 1 out of k ℓ-bit strings, e.g., [2, 21, 24]. Note, a symmetric CPIR (CSPIR) is needed for our application. CSPIR of [21] achieves costs $\Theta(s \log^2 k + \ell \log k)$, where s is a possibly non-constant security parameter. However, the break-even points where the OT sublinearity brings benefit are too high. For example, [2] costs more in communication than the naive linear-in-k OT for $k \leq 2^{40}$. Further, known CPIR protocols heavily (at least linearly in k) rely on expensive public-key operations, such as, in case of [21], length-flexible additive-homomorphic encryption (LFAH) of Damgård and Jurik [4, 3].

We also mention, but do not discuss in detail, that hardware design considers circuit minimization problems as well. However, their typical goal is to minimize chip area while allowing multiple executions of the same (sub)circuit. Current state-of-the-art in applying to MPC the powerful tool chains from hardware design is producing $10-20\%$ circuit (garble table) reduction [29, 5, 6], while our targeted circuit optimizations may result in better performance ($\approx 6.1\times$ circuit reduction achieved in our experiment.)

Semi-private function SFE (SPF-SFE) [26]. As discussed in the Introduction, SPF-SFE is a convincing trade-off between efficiency and the privacy of the evaluated function. Our work on construction of container circuits corresponds to that of privately programmable blocks (PPB) of [26], which were hand-optimized in that work. In our view, the main contribution of [26] is in identifying and motivating the problem of SPF-SFE and building a framework capable of integrating PPBs into a complete solutions. They provide a number of very simple (but nevertheless useful) PPBs. In our notation,

they consider the following sets for $\mathcal{S}$-universal circuit: $\mathcal{S}_{COMP} = \{<, >, \leq, \geq, \neq\}, \mathcal{S}_{ADD,SUB} = \{+, -\}$, $\mathcal{S}_{MULT} = \{\text{input} * \text{constant}\}$, $\mathcal{S}_{BOOLGATE} = \{\vee, \wedge, \oplus, NAND, NOR, XNOR\}$, $\mathcal{S}_{UC} = \{\text{all circuits}\}$, as well as the following sets recast from [19]: $\mathcal{S}_{SEL} = \{\text{input select circuits}\}, \mathcal{S}_{IN\_PERM} = \{\text{input permute circuits}\}$, $\mathcal{S}_{SEL} = \{\text{Y bit selector}\}, \mathcal{S}_{SEL} = \{\text{X bit selector}\}$. Each of these sets only consists of functions with already identical or near-identical topology; this is what enabled hand-optimization and optimal sizes of the containers. Other than the universal circuit PPB, no attempt was made to investigate construction PPBs of circuits of *a priori* differing topology.

In contrast, we can work with *any set* $\mathcal{S}$ of circuits for $\mathcal{S}$-universal circuit and heuristically improve on the full universal circuit, and on the standard option of evaluating of all $\mathcal{S}$ circuits and selecting the output.

GMW for multi-input gates. In independent and concurrent work, Dessouky et al. [6] discovered the same method of obtaining cheap GMW gates with multi-valued inputs by using the OT extension of [16] (multiple boolean inputs and multi-valued inputs are easily interchangeable due to [16]). In their work, Dessouky et al. make several performance optimizations to the usage of [16]. They also show in detail that for some functions, (e.g., AES), multi-input GMW gates are advantageous. In their notation, this approach is called lookup-table (LUT)-based secure computation. Our work focuses on different application of LUT-based computation, circuit clause overlay, and may achieve, in its domain, higher performance improvement.

1.4 Notation

Let f be the function we want to evaluate and C a boolean circuit representing f. We consider a `switch` statement inside f, evaluating one of k clauses depending on the internal variable or input of f. Let $C_1, ..., C_k$ be the subcircuits of C corresponding to the k clauses of f. We will often use the terms "clause" and "subcircuit" interchangeably, and their meaning will be clear from the context. For simplicity we will often discuss clauses of the same size n, although in the evaluation section we consider concrete examples with different clause sizes.

We define directed acyclic graphs (DAGs) $D_1, ..., D_k$ from circuits $C_1, ..., C_k$ where, with the exception of auxiliary nodes representing circuit inputs and outputs, the graph's nodes represent circuit gates and the graph's directed edges represent circuit wires. These graphs represent the *topology* or the *wiring* of the corresponding circuits. When the meaning is obvious from context we may interchangeably refer to these graphs/circuits as D_i or C_i. From DAGs $D_1, ..., D_k$ we will build a *container DAG* D_0, with the property that any of $C_1, ..., C_k$ can be implemented from D_0 by assigning corresponding gate functions to the nodes of D_0. We will usually call this *programming* of D_0. We note that for efficiency we may produce partially programmed D_0, i.e. one where some of the gates are already fixed. We will interchangeably refer to this container graph/circuit as D_0 and C_0. Circuits C_0 are, of course, generated for circuit-based secure computation. We specifically discuss GC and GMW protocols. We will often

unify our references to the use of GC and GMW. For example, when clear from the context, by "garbling C_0" we will mean using C_0 in either GC or GMW.

Other standard variables we will use are s, which is the computational security parameter, and n_0, which is the size of D_0. Circuits C_0 will then be evaluated In the GC protocols there are two players, GC constructor, which we will denote P1, and GC evaluator, or P2.

2 Technical Solution Overview

In this section, our goal is to describe the complete intuition behind our approach. Having this big-picture view should help put in perspective the formalizations and details that follow in the next sections.

Consider the SFE of a circuit C, and inside it a `switch` statement with k clauses/subcircuits $C_1, ..., C_k$, only one of which is evaluated based on a player's input or an internal variable. In this overview we focus on the more complex and more general second scenario (internal variable), while pointing out the very efficient solution to the first scenario as well.

Our starting point is the widely known observation that in some GC variants (e.g. in classical Yao [33, 20]), the evaluator will not learn the logic of any gate, but only the structure of the wiring of the circuit. We start by supposing that all our subcircuits already have the same wiring, i.e. the underlying DAGs are the same. We provide intuition on how to unify the wiring in the following Section 2.3.

2.1 Improved GC for `switch` of Identically-Wired Clauses

If all k clauses/subcircuits had the same topology/wiring, all that is needed is for the circuit generator to generate and deliver to the evaluator the garbling of the right subcircuit.

SPF-SFE. In the important special case where `switch` clause is selected by a player's private input, this is trivial and has no extra overhead: this player will be the GC generator and he simply sends the set of garbled tables programming the clause which corresponds to his input.

General case. Consider the case where `switch` is selected by an internal variable. One natural way to deliver the garbling would be to execute a 1-out-of-k OT on the clauses. Unfortunately, this, under the hood, would require sending garblings of each of $C_1, ..., C_k$ to the evaluator[1], which would not improve over the standard GC.

We can do better. To sketch the main idea, we let each C_i be a $\{\vee, \wedge, \oplus\}$-circuit[2]. (As all C_i are identically wired, their DAG representations D_i are the same, and the container DAG D_0 is equal to D_i. Recall, in our notation, $|D_0|=$

[1] See related work in Section 1.3 for discussion on the high costs of sublinear PIR for smaller-size DBs.

[2] We could reduce the set of considered circuit gates, e.g., by eliminating the OR gate and implementing it with AND and XOR gates. This would present us with

n_0.) For now do not consider Free-XOR; it will be clear later that our approach works with Free-XOR. Now, enumerate the gates in each C_i and let d_i be a string of length n_0 defining the sequence of gates in C_i (in our construction, each symbol in d_i will denote one of a five possible gates – $\{\vee, \wedge, \oplus\}$, as well as an auxiliary left and right input wire pass-through gates L and R). Perform 1-out-of-k OT on the strings d_i to deliver to the evaluator the right circuit definition string. Then for each gate, the players will run 1-out-of-5 OT, where the generator's input will be the five possible gate garblings, and the evaluator will use the previously obtained d_i to determine its OT choices.

Notice that each string d_i reveals to the evaluator precisely which circuit has been transferred. This is easy to hide: for each gate g_j, the GC constructor selects a random permutation π_j on the five types of gates and applies π_j to the j-th symbol of d_i during d_i construction. He also applies π_j to permute his OT input of five garbled tables. Finally, sending to the evaluator d_i based on the internal state is easy. For a `switch` with two clauses, the generator simply sends d_1 encrypted with the 0-key of the selection wire, and d_2 encrypted with the corresponding 1-key. For a `switch` with k clauses, each string d_i will be encrypted with the key derived from the wire labels corresponding to the choice of the i-th clause.

For the reader familiar with the details of standard GC, it should be clear that the above `switch`-evaluation algorithm can be readily plugged into the standard GC protocol. Let s be the computational security parameter. Following cost calculations presented after Theorem 1 and in Section 8, the communication cost of evaluating the `switch` on the k clauses will be approximately $3kn_0 + 22n_0s$. In contrast, standard GC would require sending all k garblings at the cost of $4ns$ ($2ns$ using recent half-gate garbling [34]), where $n = \sum_i |C_i|$. The $4ns$ term is the most expensive term; reducing it to $22n_0s$ and making it independent of n is the contribution of our GC protocol. We again stress that if clause is selected by GC generator, we can use all GC optimizations, and our GC cost is $2n_0s$. Finally, we note that in above calculations we did not account for the cost of circuitry selecting the output of the right clause and ignoring outputs of other clauses. This circuit is linear in ko, where o is the number of outputs in each clause. This circuit needs to be evaluated in the state-of-the-art GC, but not in our solution.

We further note that `switch` clauses can be nested. We discuss this in Section 4.

2.2 Improved GMW for `switch` of Identically-Wired Clauses

An approach similar to the one described above in Section 2.1 can be particularly efficiently applied in the GMW setting. We will take advantage of our novel observation on the cost of multi-input GMW gates under the OT extension of Kolesnikov and Kumaresan [16].

the trade off between more efficient gate processing and potentially larger circuits. Alternatively, we could consider a larger gate basis and have costlier gate processing. We defer exploration of these trade-offs as future work.

As in our GC protocol above, we consider the circuit definition strings d_i. As in the GC protocol, for each gate g_j, one player selects a random permutation (or mask) π_j on the five types of gates and applies π_j to the j-th symbol of d_i during d_i construction. This masked definition string is transferred to the other player via OT.

In contrast with GC, we will not do the expensive 1-out of-5 OT on garbled gates. In GMW, we will evaluate gates on three input wires: two circuit wires and one 5-valued wire selecting the gate function ($\{\vee, \wedge, \oplus, L, R\}$). The players thus will run 1-out of-20 OT (the 20 possibilities are the five gate functions, each with four wire input possibilities) to obtain the secret share of the output.

Our simple but critical observation is that with using [16] OT, and because the GMW secret shares are a single bit each, the evaluation of multi-input gate, for moderate number of inputs, *costs approximately the same* as that of the two-input gate. Indeed, the main cost of the OT is the [16] rows transfer. Sending the encryptions of the actual secrets, while exponential in the number of inputs, is dominated by the OT matrix row transfer for gates with up to about 8 binary inputs. In our case, sending of 20 secrets requires only 20 bits (one bit per secret) in addition to the OT matrix transfer. Thus, additional communication as compared to standard 1-out of-4 GMW OT extension (also implemented via [16]) is only $20 - 4 = 16$ bits!

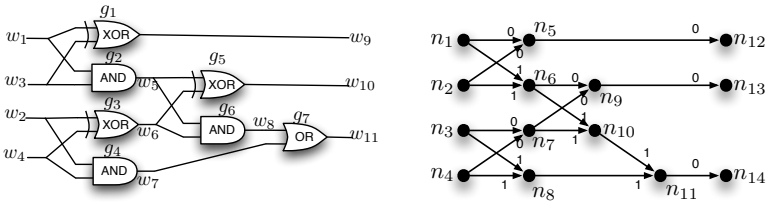
As a result, the circuit reduction achieved by embedding several clauses into one container is directly translated into the overall improvement for semi-honest GMW protocol.

2.3 Efficient Circuit Embedding to Obtain Identically-Wired Clauses

We now describe the intuition behind our graph/circuit embedding algorithm, as well as summarize its performance in terms of the size of the embedding graph. In Section 3, we describe a circuit embedding algorithm, which takes as input the set of k circuits $C_1, ..., C_k$ and returns an (unprogrammed) container circuit C_0 capable of embedding each of these circuits, as well as the programming strings needed to generate the garblings of C_0 which implement/garble each C_i.

Our approach is graph theoretic. Assume for simplicity that we have exactly two input circuits. As a first step, we translate each circuit C_i to a directed acyclic graph (DAG) D_i (see Figure 1 for example and Section 3 for a formal definition). The problem of finding a "small" container circuit embedding both C_1 and C_2 is now reduced to finding a "small" DAG which "contains" D_1 and D_2. Informally, a DAG D 'contains' another DAG D' if through a series of node deletions, edge deletions and replacing each 3-node path uvw where v has in-degree and out-degree 1 with a 2-node path, i.e., an edge, uw in D, one can recover a graph isomorphic to D'.

We start by showing that if the input DAGs are restricted to have out-degree at most one, then there exists a polynomial time algorithm (Algorithm 1) to find a DAG D_0, also of out-degree at most one, of minimum size. We remark that our approach is closely related to the classical polynomial time algorithm for



(a) C with wires $w_1, ..., w_{11}$ and gates $g_1, ..., g_7$.

(b) D with nodes $n_1, ...n_{14}$ corresponding to C's gates, inputs and outputs.

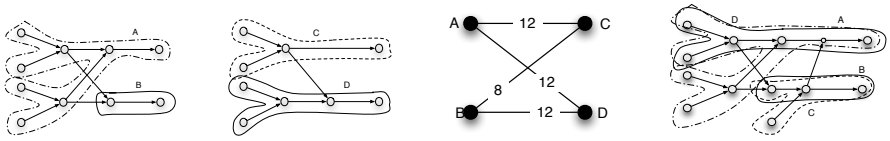
Fig. 1. A 2-bit adder circuit C and corresponding circuit DAG D. Edges of D are assigned weights to facilitate Free XOR optimizations (cf. Sect. 3).

testing whether or not two trees are isomorphic [31], though it is more difficult than this. Indeed, the operation which replaces 3-node paths with 2-node paths is closely related to edge contraction of graph minors [28].

Restricting DAGs to having out-degree at most one corresponds to restricting circuits to having fan out at most one and is, of course, unrealistic. To develop a general algorithm (see Figure 2 for a toy example), we observe that the nodes of every DAG D with r sinks can be covered by a set of r DAGs each with out-degree at most one, i.e. *subtrees*. For each pair of such subtrees (one from D_1 and one from D_2) we first apply Algorithm 1 to determine the minimum cost (roughly, the minimum $|D_0|$) of co-embedding the pair. We use these costs to weight an auxiliary complete bipartite graph: roughly, one part is labeled by the subtrees of D_1, one part is labeled by the subtrees of D_2, and the weight of the edge is the minimum cost of co-embedding the subtrees corresponding to the edge's endpoints. The minimum weight perfect matching in this graph corresponds to a valid container circuit that can be easily constructed. In generality, only considering subtrees covering the nodes of D_1, D_2 may leave out some edges, which we then appropriately reinsert into D_0 to guarantee that D_0 will be universal for both D_1, D_2.

We now turn to the performance of our algorithm. Clearly any circuit embedding of circuits C_1 and C_2 has size at least $\max\{|C_1|, |C_2|\}$ and needs to have size at most $|C_1|+|C_2|$[3]. Our experimental validation (see Section 8) embeds two circuits into a circuit whose size is on average 15.1 percent of the way between these trivial lower and upper bounds. *Assuming this embedding performance*, by divide-and-conquer repeated embedding we would obtain an embedding of k circuits of size n into a circuit of size $1.151^{\log k} n = k^{0.203} n$ (Lemma 1).

[3] For some pairs of circuits our heuristic may produce a container circuit of size greater than $|C_1|+|C_2|$. In this case, we will simply take C_0 to include both C_1 and C_2.



(a) D_1 with A and B subtrees.

(b) D_2 with C and D subtrees.

(c) Pairwise cost of co-embedding subtrees.

(d) Low cost D_0 embedding both D_1 and D_2.

Fig. 2. Determining a low cost (circuit) DAG embedding two input (circuit) DAGs.

2.4 NP-hardness of Graph Embedding

In Section 7 we show that the problem of finding a minimum-cost circuit C_0 into which two given circuits C_1 and C_2 can be embedded is NP-complete. The proof uses a reduction from the well-known NP-complete problem 3-SAT [10]. In fact, the reduction shows the somewhat stronger result that says that the problem remains NP-complete even when one of C_1 or C_2 is a tree (and the other a DAG) and both have bounded in-degree and out-degree.

Intuitively, the idea of the reduction is that the DAG, say C_1, represents all possible truth assignments of the variables and all possible ways to satisfy each clause in the 3-SAT instance while the tree C_2 represents the requirement that each variable must be set to true or false and the requirement that each clause has (at least) one resulting literal that is true. Then we show that an embedding of C_2 into C_1 is possible if and only if there is a satisfying assignment for the 3-SAT instance. Clearly such an embedding has minimum cost. When such an embedding exists, it can easily be interpreted as a particular truth assignment to the variables and an "assignment" of one resulting true literal to each clause.

3 Definition of Circuit Embedding

In this section, we bridge circuit-based SFE and graph theory. In particular, we describe the CIRCUIT EMBEDDING ALGORITHM, which takes as input the set of k circuits $C_1, ..., C_k$ and returns the desired n_0-gate *container circuit* C_0 together with *definition (programming) strings* $d_1, ..., d_k$. Specifically, a container circuit is an unprogrammed circuit, that is, a collection of gates each of whose function is unspecified, though the wire connections between these gates are fixed. The function of these gates is then specified by choosing a programming stream, which is a mapping from the gates to the functions $\{\vee, \wedge, \oplus, L, R\}$, where L (resp. R) is the left (resp. right) wire pass through gate. To describe the CIRCUIT EMBEDDING ALGORITHM, we start by describing a mapping between circuits and a specific type of weighted directed acyclic graphs (DAGs) and the graph theoretic equivalent of the Garbled Circuit approach we are proposing.

Let C be a circuit defined by gates $g_1, ..., g_n$ and wires $w_1, ..., w_m$. We use the following weighted directed acyclic graph (DAG) $D = (V, A, w)$ to represent

it. The node set V has three parts: for each wire w_i that is an input to C we add an "input" node n_i, for each output wire w_i, we add an "output" node n_i, and for each gate g_i, we introduce a "gate" node n_i. All directed edges in E are directed in the direction of evaluation. Specifically, for each input wire to gate g_i there is an edge from its corresponding "input" node to the "gate" node n_i. For each output wire from gate g_i there is an edge from the "gate" node n_i to its corresponding "output" node. For each wire from gate g_i to gate g_j, there is an edge from n_i to n_j. Finally, for simplicity in dealing with free-XORs and the cost of circuit, we give each edge a weight. For a gate node g_i corresponding to an XOR-gate, we give all in-edges e of g_i weight $w_e = 0$; for output nodes n_i, we give all in-edges e of n_i weight $w_e = 0$; for all other edges e receive weight $w_e = 1$. See Figure 1 for an example. We call such a DAG, the circuit DAG. We remark that given a circuit DAG we can always determine an unprogrammed circuit corresponding to it.

The cost of a circuit is the total size of the truth tables needed to represent it, i.e., $\sum_{\text{non-XOR } g_i} 2^{\{\text{fan in of gate } g_i\}}$, where XOR-gates add zero cost [18]. This translates to the corresponding circuit DAG as $\text{cost(D)} := \sum_{\text{u}\in\text{D}} 2^{\sum_{\text{v}\in\text{N}_{\text{D}}^{-}(\text{u})} \text{w}_{\text{vu}}}$, where $N_D^-(u)$ is the set of in-neighbors of node $d \in D$.

We are interested in the minimum cost container circuit C_0 that can be used to embed circuits $C_1, ..., C_k$. Necessarily, this requires that for each C_j there is a 1-1 mapping f from the gates of C_i to C_0, such that, for each wire of C_j between gate g_i and $g_{i'}$ there is a set of wires linking $f(g_i)$ and $f(g_{i'})$. Moreover and as we now describe, the flow of information of C_j must be preserved in C_0.

An *out-arborescence* is a directed acyclic graph that is weakly connected[4] and every node has in-degree at most one. We define the *source* of an out-arborescence T, denoted source(T), as the unique vertex with in-degree zero. Let $D' = (V', A', w')$ and $D = (V, A, w)$ be DAGs.

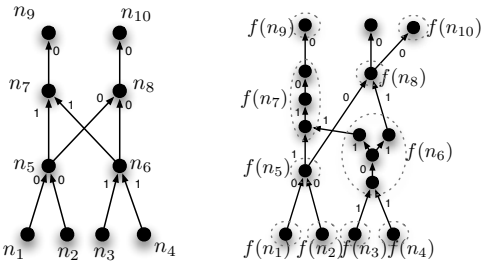
Definition 1. An *embedding* of D' into D is a mapping f from **nodes** of V' to **out-arborescences of** D and from **(weighted) directed-edges of** A' to **(weighted) directed-edges of** A satisfying

1. for all $u' \neq v' \in V'$, $f(u') \cap f(v') = \emptyset$,
2. for $u'v' = e' \in A'$, $\exists x \in f(u')$ such that $f(e')$ starts at x and ends at the source of $f(v')$, and
3. for $u'v' = e' \in A'$, $w'_{e'} \leq w_{f(e')}$.

It follows immediately from the definition that there is a 1-1 mapping between nodes of D' and the sources of the out-arborescences in D specified by f. Moreover, for every node n' of D' and source of $f(n') = n$, f is a mapping such that for each in-edge e' of n' and there is a unique in-edge $e = f(e')$ of n such that $w'_{e'} \leq w_e$. From this it follows that the sum of the weights on the in-edges of n is at least as large as the sum of the weights on the in-edges of n'. Hence, we have the following observation.

[4] A directed graph is weakly connected if replacing all edges with undirected edges yields a connected graph, that is, every pair of nodes in the graph is connected by some path.

Observation 1 $\text{cost(D)} \geq \text{cost(D}')$.



(a) D' (b) D

Fig. 3. An example embedding f of D' in D.

We now are in a position to describe the Circuit Embedding Algorithm. Let $C_1, ..., C_k$ be the set of k-input circuits. First, we find the corresponding circuits DAGs $D_1, ..., D_k$. Second, given this set of circuits DAGs, we determine a *low cost* circuit DAG D_0 that embeds each of $D_1, ..., D_k$ with functions $f_1, ..., f_k$. The heuristic we describe in Section 6 is one approach to solve this second step. Third, we determine the container circuit C_0 as the circuit corresponding to D_0. Finally, we determine the programming string d_i for each i. To do so, we need only specify the function of each gate node in D_0. For a specific embedding f_i, each gate node v of D_0 is either A) a source or B) a non-source node of some out-arborescence. In the former case, $d_i(v)$ is equal to either AND, OR or XOR depending on the function of the pre-image of the out-arborescence rooted at v. In the latter case, $d_i(v)$ is equal to L as the left input wire pass-through.

4 GC Protocol for Overlaying Subcircuits

In this section we will formalize the intuition of Section 2.1. Namely, we will present a full GC protocol with processing of k *identically wired* `switch` clauses at approximately the cost of *one* such clause, and prove its security. Of course, identically wired clauses are not typical in circuits. In Section 6 we show how to embed a number of arbitrary circuits into a single container circuit, so that each of the circuits could be implemented by a corresponding programming of the gates of the container circuit. Our approach is heuristic, but we of course can always fall back on running standard GC/GMW protocols if the performance of the heuristic does not outperform.

Our approach can be instantiated using a number of GC garbling techniques. For simplicity of presentation, and because it is a standard GC trick, in the following presentation we omit assigning and processing the wire key pointers which will tell the evaluator which garbled table row to decrypt. Also, we do not

include Free-XOR in this algorithm. We will argue later that our construction allows to take full advantage of Free-XOR. Finally, for ease of presentation and w.l.o.g., our construction is for functions with a single `switch`.

Consider an $\{\vee, \wedge, \oplus\}$ circuit C with a `switch` $(C_1, ..., C_k)$ statement, which evaluates one of subcircuit clauses $C_1, ..., C_k$ based on an internal variable. Let Enc, Dec be a semantically secure encryption scheme.

Protocol 1 *(GC with* `switch` *statements)*

1. **Once-per-function Precomputation.** *Parse C, identify* `switch` $(C_1, ..., C_k)$*, and call the graph embedding algorithm on $C_1, ..., C_k$. Obtain the container circuit C_0 of size n_0 as well as k circuit programming strings $d_1, ..., d_k$, each of size n_0. Each d_i will consist of symbols $\{\vee, \wedge, \oplus, L, R\}$, where L (resp. R) is the left (resp. right) input wire pass-through gate. Denote the j-th symbol of d_i by $d_{i,j}$. Let C' be the C with the* `switch` $(C_1, ..., C_k)$ *replaced with C_0. C' is assumed known to both players before the computation.*

2. *For each wire W_i of C', GC generator randomly generates two wire keys w_i^0, w_i^1.*
3. *For each gate g_i of $C' \setminus C_0$ in topological order, GC generator garbles g_i to obtain the garbled gate table. For each of 2^2 possible combinations of g_i's input values $v_a, v_b \in \{0, 1\}$, set*

$$e_{v_a,v_b} = H(k_a^{v_a} || k_b^{v_b} || i) \oplus w_c^{g_i(v_a,v_b)}$$

   Garbled table of g_i is a randomly permuted set $\{e_{v_a,v_b}\}, v_a, v_b \in \{0, 1\}$.
4. *GC generator sends all generated garbled tables to GC evaluator. Garblings of inputs of C' are sent to GC evaluator directly and via OT, as is standard in GC.*
5. *GC generator generates n_0 random permutations π_i over $\{\vee, \wedge, \oplus, L, R\}$.*
6. *GC generator computes the following. Let $W_{j_1}, ...W_{j_t}$ be the wires defining the* `switch` *choice, $t = \lceil \log k \rceil$. For $i = 1$ to k, set $\tilde{d}_i = \pi_1(d_{i,1}), ..., \pi_{n_0}(d_{i,n_0})$. Now, each $\tilde{d}_i$ looks random as an independent random permutation π_j was applied to each symbol $d_{i,j}$. Let $ED = Enc_{key_1}(\tilde{d}_1), ..., Enc_{key_k}(\tilde{d}_k)$. Here key_i is derived from wire keys of $W_{j_1}, ...W_{j_t}$, corresponding to* `switch` *selection i, by setting $key_i = H(\text{“}\texttt{switchkey}\text{”}, w_{j_1}^{i_1}, ...w_{j_t}^{i_t})$.*
7. *GC generator sends encrypted circuit definition strings ED to GC evaluator, in a random order.*
8. *GC evaluator evaluates in topological order all gates that are possible[5]; in particular, the wire keys defining the values of the* `switch` *statement will be known to GC evaluator.*
9. *GC evaluator derives the decryption key for $Enc_{key_i}(\tilde{d}_i)$ and decrypts to obtain $\tilde{d}_i$, the (permuted) definition string for the clause to be evaluated. Evaluator will know which string to decrypt by including an additional pointer bit in the wire labels of $W_{j_1}, ...W_{j_t}$ (point-and-permute).*

[5] Recall, for simplicity we did not explicitly include the standard permute-and-point table row pointers in our protocol. We assume the evaluator knows decryption of which row to use, e.g. via using the standard permute-and-point technique.

10. *For each gate $g_i \in C_0$, in topological order*
    (a) *GC generator prepares five garbled tables, $\{T_\vee, T_\wedge, T_\oplus, T_L, T_R\}$ implementing one each of gate functions $\{\vee, \wedge, \oplus, L, R\}$, i.e. OR, AND, XOR, Left wire pass-through, Right wire pass-through. Note that all five garbled tables are constructed with respect to the same input/output wire labels of gate g_i.*
    (b) *The two players execute in parallel n_0 semi-honest 1-out-of-5 OT protocols, for j from 1 to n_0. Here GC generator's input is $\pi_j(\{T_\vee, T_\wedge, T_\oplus, T_L, T_R\})$, and GC evaluator's input is the symbol of the programming string obtained in Step 9, i.e. $\pi_j(\{\vee, \wedge, \oplus, L, R\})$. As a result, GC evaluator receives garbled gate tables of the remaining gates.*
11. *GC evaluator evaluates in topological order all remaining gates of C' and sends output wire keys to generator for decryption.*

Observation 2 *For simplicity of presentation and to focus on the novel contribution, we omitted explicitly writing out some standard GC techniques, such as permute-and-point.*

Observation 3 (Free-XOR compatibility) *We presented the protocol without regard to free-XOR. However, it is easy to see that our construction is compatible with it. Indeed, as is also argued in discussion on the circuit embedding heuristic in Section 6, the generated container circuit will have many gates fixed to be XOR gates, rather than placeholders for one of $\{\vee, \wedge, \oplus, L, R\}$. It is easy to see that since any of k clauses could be implemented in the container circuit, and "permanently" fixing some of its gates to be XOR is done in circuit pre-processing, this will not affect security.*

In our circuit embedding heuristics, we aimed to maximize the number of such gates so as to take the full advantage of free-XOR.

We note that it is not immediately clear how to use the 3-row garbled-row reduction (GRR3) of [27] in our approach. This is because the GRR3 idea is to define one of the garbled rows as a function of garbled values of the two input wires which result in this output value (and omit that row from the table). However, in our setting, we don't know which gate function will be used. Hence, the 0-1 semantics of the implicitly defined gate output label may be different for different gate functions which may program this specific gate of C_0. This will cause problems for garbling subsequent gates. We thus do not use GRR3, and use standard 4-row tables. We discuss optional inclusion of NOT gates in our circuits in the full version.

Theorem 1. *Let OT protocol be secure in the semi-honest model. Let Enc be semantically-secure encryption. Let H be a hash function modeled by a random oracle. Then Protocol 1 is secure two-party computation protocol in the semi-honest model.*

The proof (with respect to the standard security definition of secure computation) is deferred to the full version, due to the lack of space.

Cost calculation. As compared to plain GC of C, our protocol uses additional OT instances. This comes cheap due to the Ishai et al.'s OT extension [14] and follow-up optimizations, such as [1, 16]. Further, an extension of [14] for 1-out of-k OT of Kolesnikov and Kumaresan [16] can be effectively used for our 1-out of-5 OTs.

In detail, let s be the computational security parameter, and take the size of each garbled table as $4s$. Then the communication cost of evaluating the `switch` on the k clauses embedded in container C_0 of size n_0 will be approximately $3kn_0 + 22n_0s$.

Indeed, 1-out of-k OT of circuit programming strings will take approximately $3kn_0$ bits (k encryptions of $3n_0$-bit long strings, plus a 1-out of-k OT on short decryption keys of size s, whose cost is small and is ignored.) Running 1-out of-5 OT on gate tables of size $4s$ is done via [16]. (Recall, [16] shows how to do 1-out of-5 OT for only double the cost of 1-out of-2 OT.) The cost consists of sending 5 encryptions each of length $4s$, and running 1-out of-4 OT on random secrets of size s, which costs approximately $2s$, i.e. one OT extension matrix row of [16]. Summing up, we get our cost approximately $3kn_0 + 22n_0s$.

Ignoring lower order term $3kn_0$, *we can view our communication cost per gate as approximately factor* 5.5 *of that of the standard Yao-gate*, and factor 11 of that of the optimal garbling of Zahur et al. [34]. We note that in cases where clause is selected by the input of a player (GC generator), our cost of each gate is the same as that of [34]. We finally note that we, in contrast with all prior GC protocols, *do not* need to include the circuitry selecting the output of the right clause and ignoring outputs of other clauses.

We discuss experimental results, which depend on the quality of embedding, in Section 8.

Nesting `switch` statements

We observe that a natural implementation of `switch` nesting will be secure and cheap. Intuitively, this is because the vast majority of the cost – OTs of the gates – will remain unaffected by sub-switches, and *only the programming strings management will need to be adjusted*.

Due to the lack of space, we present the detailed nesting construction in the full version.

5 GMW Protocol for Overlaying Subcircuits

Our GMW protocol is a natural recasting of our GC protocol into the GMW approach, with the exception of us making and exploiting the novel observation that multi-input gates in GMW are cheap. In GMW, we will program a gate simply by viewing it as having an additional 5-ary function definition input. The circuit programming string will be secret-shared among the players with a $(2, 2)$ secret sharing, just like regular GMW wire values. Thus our programmable gate evaluation is just a slight generalization of the GMW evaluation.

Protocol 2 *(GMW with* `switch` *statements, sketch)*

1. **Once-per-function Precomputation.** *Parse C, identify* `switch` $(C_1, ..., C_k)$*, and call the graph embedding algorithm on $C_1, ..., C_k$. Obtain the container circuit C_0 of size n_0 as well as k circuit programming strings $d_1, ..., d_k$, each of size n_0. Each d_i will consist of symbols $\{\vee, \wedge, \oplus, L, R\}$, where L (resp. R) is the left (resp. right) input wire pass-through gate. Denote the j-th symbol of d_i by $d_{i,j}$. Let C' be the C with the* `switch` $(C_1, ..., C_k)$ *replaced with C_0. C' is assumed known to both players before the computation.*

2. *Beginning with the secret sharing of the inputs, for each gate g_i of $C' \setminus C_0$ in topological order, players evaluate the gates according to the GMW protocol. In particular, wires defining the the* `switch` *choice will be processed.*
3. *Let $W_{j_1}, ...W_{j_t}$ be the wires defining the* `switch` *choice, $t = \lceil \log k \rceil$. Players use OT to generate a $(2, 2)$ secret sharing of the selected programming string as follows.*
   (a) *GC generator generates n_0 random permutations $\pi = \{\pi_j\}$ over $\{\vee, \wedge, \oplus, L, R\}$.*
   (b) *GC generator computes the following. For $i = 1$ to k, set $\tilde{d}_i = \pi_1(d_{i,1}), ..., \pi_{n_0}(d_{i,n_0})$. Now, each $\tilde{d}_i$ looks random as an independent random permutation π_j was applied to each symbol $d_{i,j}$.*

   Player P_2 uses his shares of $W_{j_1}, ...W_{j_t}$ to obtain (via OT with P_1) the permuted programming string $\tilde{d}_i$.
4. *Players proceed to evaluate all remaining gates of C'. The gates in C_0 have an additional input specifying the gate function. This input is taken from the circuit programming string d. Note that d is already secret-shared among the two players: P_1 has π, P_2 has $\tilde{d}_i$. Each (two-input Boolean) gate of C_0 is evaluated by a slight generalization of GMW, where 1-out of-$4 \cdot 5$ OT is run by the players.*
   Specifically, for each of the five possible gate functions $\{\vee, \wedge, \oplus, L, R\}$ for gate $g_j \in C_0$, P_1 prepares four corresponding GMW OT secrets. Then P_1 permutes the five groups of four GMW OT secrets according to π_j. Then P_1 and P_2 run 1-out of-$4 \cdot 5$ OT, where P_2's input is $\tilde{d}_{i,j}$ and the GMW shares of the wire values.
5. *Players combine their shares on the output wires of C' and reconstruct the output.*

Theorem 2. *Let OT protocol be secure in the semi-honest model. Then Protocol 2 is a secure two-party computation protocol in the semi-honest model.*

Proof. The proof of security of this protocol in the semi-honest model is trivial. Indeed, assuming the security of OT, it is easy to check that players ever receive only secret shares of the wire values. We omit this simple proof.

Cost calculation. As compared to standard GMW protocol, our protocol requires OT of the programming strings. Additionally, it evaluates multi-input

gates, resulting in 1-out of-20 OT. As discussed above, in particular in GC cost calculation, the OT of programming strings is a low-order cost term and can be ignored. Further, as explained in detail in Section 2.2, the cost of 1-out of-20 OT using [16] is only 16 bits greater than that of the 1-out of-4 OT, and hence this difference can also be swept under the rug. We conclude that the above Protocol 2 (GMW with `switch` statements) implements the oblivious circuit programming at the cost similar to that of the standard GMW protocol.

Nesting `switch` statements. The discussion and results of GC-based nesting (Section 4) directly applies to the GMW setting.

6 Embedding Circuits of Bounded Fan In

In this section, we sketch a heuristic algorithm which given a set of k circuit DAGs, $D_1, ..., D_k$, returns a circuit DAG D_0 such that for each D_i there exists an embedding f_i into D_0, and D_0 has as small of cost as possible. We proceed in two main steps.

First, in Section 6.1, we restrict our attention to circuits that have fan out one and fan in bounded by 2, though a straightforward generalization leads to fan in bounded by any constant. These are commonly referred to as *in-arborescences of bounded in-degree* 2, but for ease of exposition we will call them *tree circuits*. We describe a polynomial time **exact** algorithm that given two circuit trees T_1 and T_2 finds a circuit tree T of minimum cost embedding both T_1 and T_2. Specifically, we prove the following.

Definition 2. The *cost* of embedding a set of circuit DAGs $D_1, ..., D_k$, denoted $\mathrm{cost}(\mathrm{D}_1, ..., \mathrm{D}_\mathrm{k})$, is the cost of a circuit DAG D_0 of minimum cost such that there is an embedding of D_i into D_0 for all $i = 1..k$.

Theorem 3. *Let T_1 and T_2 be tree circuits. There exists an $O(|T_1||T_2|)$ algorithm to determine an optimal, i.e. minimum cost, tree circuit T embedding both T_1 and T_2.*

Second, in Section 6.2, we remove the bound on the fan out, only requiring that the input circuits have fan in bounded by 2. We describe an algorithm which relies on the algorithm of Section 6.1 as a subroutine. Letting D_1 and D_2 be circuit DAGs of fan out 2, we describe a polynomial time heuristic algorithm to a determine circuit DAG D_0 embedding both D_1 and D_2.

A straightforward approach, using this heuristic as a subroutine, then allows for k-circuit inputs. The following lemma describes an algorithm which returns a circuit whose size grows sublinearly in the number of input circuits (assuming certain performance of the heuristic). Because we condition it on the heuristic, we stress that the Lemma cannot be universally applied. Rather, we use the Lemma to formalize the embedding algorithm for larger number of clauses, as well as to discuss its projected performance.

Lemma 1. *Let $\tau > 1$. Assume there exists an algorithm which takes as input circuit DAGs D' and D'', each of size exactly n, and returns a circuit DAG*

D_0 embedding both D' and D'' whose size is at most τn. Then there exists an algorithm which takes as input circuit DAGs $D_1, ..., D_k$, each of size exactly n, and returns D_0 of size at most $k^{\log_2 \tau} n$.

Proof. Let $D_1, .., D_k$ be a set of circuit DAGs and assume $k = 2^\ell$. We first apply the heuristic of Section 6 to determine a circuit DAGs D_i^2 embedding both D_{2i-1} and D_{2i} for each $i = 1, ..., k/2$. Iterating this for $j \geq 2$, for each $i = 1, ..., k/2^j$ we determine D_i^j from D_{2i-1}^{j+1} and D_{2i}^{j+1}. We then return $D = D_1^\ell$.

We prove the size bound by induction, where the base case is assumed. By induction, assume that the algorithm returns D_i^{j-1}, $i = 1, ..., k/2^j$, each of whose size is at most $\tau^{j-1} n$. Hence, the size of $D_{i'}^j$ is at most $(\tau) * \tau^{j-1} n = \tau^j n$. Setting $j = \ell = \log k$ yields the desired result. □

In Section 8, we evaluate the performance of the heuristic, which, in our experiments on average achieves a τ value of 1.151. Assuming this τ value, Lemma 1 would imply the k-circuit size is at most $k^{0.203} \times \max\{|D_1|, |D_2|, ..., |D_k|\}$.

6.1 Tree Circuits

In order to prove Theorem 3, we use dynamic programming and *match* pairs of vertices of T_1 and T_2 as follows. For simplicity,we omit dealing with Free-XOR for now. Let $\delta^-(v)$ be the in-degree of a node.

Definition 3. For circuit DAG D and $t \in D$, let $D[t]$ be the circuit DAG induced on vertices v such that there exists a directed path from from v to t in D.

Definition 4. Define the *matchcost* of $a \in T_1$ and $b \in T_2$ as the minimum cost of a tree T such that there exists a mapping f_1 that embeds $T_1[a]$ into T and a mapping f_2 that embeds $T_2[b]$ into T where $f_1(a) = f_2(b)$. Denote this minimum cost by $\mathrm{match(a, b)}$.

Consider computing $\mathrm{cost(T_1, T_2)}$ where a is the root of T_1 and b is the root of T_2. Clearly, there is no advantage, with respect to cost, to mapping a and b to disjoint subtrees of T and so either (i) $f_1(a) \in T[f_2(b)]$, or (ii) $f_2(b) \in T[f_1(a)]$. From this it follows that we can compute $\mathrm{cost(T_1, T_2)}$ by considering $O(|T_1|+|T_2|)$ matchcosts.

Definition 5. Let T_1 and T_2 be tree circuits with roots a and b, respectively. Define:

(i) $\mathrm{cost_2(T_1, T_2)} := \min_{\mathrm{t} \in \mathrm{T_2}} (\mathrm{cost(T_2) - cost(T_2[t]) + match(a, t)})$.
(ii) $\mathrm{cost_1(T_1, T_2)} := \min_{\mathrm{t} \in \mathrm{T_1}} (\mathrm{cost(T_1) - cost(T_1[t]) + match(t, b)})$.

Lemma 2. *Let T_1 and T_2 be tree circuits with roots a and b, respectively. Let T be a minimum cost tree circuit with f_1 embedding T_1 and f_2 embedding T_2.*

(i) If $f_1(a) \in T[f_2(b)]$, then $\mathrm{cost(T_1, T_2) = cost_2(T_1, T_2)}$,
(ii) If $f_2(b) \in T[f_1(a)]$, then $\mathrm{cost(T_1, T_2) = cost_1(T_1, T_2)}$.

Proof. Without loss of generality, assume that $f_1(a) = t' \in T[f_2(b)]$ and consider the minimum cost and minimum edge tree circuit T. The root r of T is equal to $f_2(b)$ (by minimality) and there exists $t \in T_2$ such that $f_2(t) = t'$. We have that $\mathrm{cost}(\mathrm{T}_1, \mathrm{T}_2)$ is equal to the cost of embedding the tree $T_2 - T_2[t]$ plus the minimum cost of a tree T' that embeds both $T_2[t]$ and T_1 given that a and t are mapped to the root of T'. Hence, $\mathrm{cost}(\mathrm{T}_1, \mathrm{T}_2) = \mathrm{cost}(\mathrm{T}_2 - \mathrm{T}_2[\mathrm{t}]) + \mathrm{match}(\mathrm{a}, \mathrm{t}) = \mathrm{cost}(\mathrm{T}_2) - \mathrm{cost}(\mathrm{T}_2[\mathrm{t}]) + \mathrm{match}(\mathrm{a}, \mathrm{t}) = \mathrm{cost}_2(\mathrm{T}_1, \mathrm{T}_2)$. The lemma follows. □

Corollary 1. $\mathrm{cost}(\mathrm{T}_1, \mathrm{T}_2)$ *is equal to the minimum of* $\mathrm{cost}_1(\mathrm{T}_1, \mathrm{T}_2)$, *and* $\mathrm{cost}_2(\mathrm{T}_1, \mathrm{T}_2)$.

In order to achieve the runtime of Theorem 3, we observe that we can determine these costs using the children of a and b together with a single match.

Lemma 3. *Let T_1 and T_2 be tree circuits with roots a and b, respectively. Then,*

$$\mathrm{cost}_1(\mathrm{T}_1, \mathrm{T}_2) = \min\Bigg\{\mathrm{match}(\mathrm{a}, \mathrm{b}), \min_{a' \in N^-_{T_1}(a)} (\mathrm{cost}(\mathrm{T}_1) - \mathrm{cost}(\mathrm{T}_1[\mathrm{a}']) + \mathrm{cost}_1(\mathrm{T}_1[\mathrm{a}'])(\mathrm{T}_2))\Bigg\},$$

$$\mathrm{cost}_2(\mathrm{T}_1, \mathrm{T}_2) = \min\Bigg\{\mathrm{match}(\mathrm{a}, \mathrm{b}), \min_{b' \in N^-_{T_2}(b)} (\mathrm{cost}(\mathrm{T}_2) - \mathrm{cost}(\mathrm{T}_2[\mathrm{b}']) + \mathrm{cost}_2(\mathrm{T}_1)(\mathrm{T}_2[\mathrm{b}']))\Bigg\}.$$

Proof. We have that

$$\begin{aligned}
&\min_{t \in T_1[a']} (\mathrm{cost}(\mathrm{T}_1) - \mathrm{cost}(\mathrm{T}_1[\mathrm{t}]) + \mathrm{match}(\mathrm{t}, \mathrm{b})) \\
&\quad = \mathrm{cost}(\mathrm{T}_1) - \mathrm{cost}(\mathrm{T}_1[\mathrm{a}']) + \min_{\mathrm{t} \in \mathrm{T}_1[\mathrm{a}']} (\mathrm{cost}(\mathrm{T}_1[\mathrm{a}'])\mathrm{cost}(\mathrm{T}_1[\mathrm{t}]) + \mathrm{match}(\mathrm{t}, \mathrm{b})) \\
&\quad = \mathrm{cost}(\mathrm{T}_1) - \mathrm{cost}(\mathrm{T}_1[\mathrm{a}']) + \mathrm{cost}_1[\mathrm{T}_1[\mathrm{a}']][\mathrm{T}_2].
\end{aligned}$$

Hence, $\mathrm{cost}_1(\mathrm{T}_1, \mathrm{T}_2)$

$$\begin{aligned}
&= \min_{t \in T_1} (\mathrm{cost}(\mathrm{T}_1) - \mathrm{cost}(\mathrm{T}_1[\mathrm{t}]) + \mathrm{match}(\mathrm{t}, \mathrm{b})) \\
&= \min\{\mathrm{match}(\mathrm{a}, \mathrm{t}), \min_{\mathrm{a}' \in \mathrm{N}^-_{\mathrm{T}_1}(\mathrm{a})} \min_{\mathrm{t} \in \mathrm{T}_1[\mathrm{a}']} (\mathrm{cost}(\mathrm{T}_1) - \mathrm{cost}(\mathrm{T}_1[\mathrm{t}]) + \mathrm{match}(\mathrm{t}, \mathrm{b}))\} \\
&= \min\{\mathrm{match}(\mathrm{a}, \mathrm{b}), \min_{\mathrm{a}' \in \mathrm{N}^-_{\mathrm{T}_1}(\mathrm{a})} (\mathrm{cost}(\mathrm{T}_1) - \mathrm{cost}(\mathrm{T}_1[\mathrm{a}']) + \mathrm{cost}_1[\mathrm{T}_1[\mathrm{a}']][\mathrm{T}_2])\},
\end{aligned}$$

completing the proof of the lemma. □

From Lemma 2, in order to determine $\mathrm{cost}(\mathrm{T}_1, \mathrm{T}_2)$, it remains to show how to determine $\mathrm{match}(\mathrm{a}, \mathrm{b})$. Since the mapping of a and b are fixed, matchcosts are easier to compute. Indeed, we can assume $f_1(a) = f_2(b)$ is the root of T. Moreover, if either $T_1[a]$ or $T_2[b]$ is a singleton then $\mathrm{match}(\mathrm{a}, \mathrm{b})$ can be determined in a straightforward way.

Observation 4 *If $T_1[a]$ is a singleton, then for all $b \in T_2$,* $\mathrm{match(a, b)} = \mathrm{cost(T_1[a], T_2[b])} = \mathrm{cost(T_2[b])}$. *If $T_2[b]$ is a singleton, then for all $a \in T_1$,* $\mathrm{match(a, b)} = \mathrm{cost(T_1[a], T_2[b])} = \mathrm{cost(T_1[a])}$.

From Observation 4 it is trivial to determine $\mathrm{match(a, b)}$ whenever either a is a leaf of T_1 or b is a leaf of T_2. Specifically, in the case that b is a leaf, we have $\mathrm{match(a, b)} = \sum_{\mathrm{t} \in \mathrm{T_1[a]}} 2^{\sum_{\mathrm{v} \in \mathrm{N^-_{T_1}(t)}} \mathrm{w_{vt}}}$ and when a is a leaf, $\mathrm{match(a, b)} = \sum_{\mathrm{t} \in \mathrm{T_2[a]}} 2^{\sum_{\mathrm{v} \in \mathrm{N^-_{T_2}(t)}} \mathrm{w_{vt}}}$.

We therefore can assume that $T_1[a]$ and $T_2[b]$ each have at least three vertices. To determine $\mathrm{match(a, b)}$ we simply consider all possible pairings of the children.

Lemma 4. *For $a \in T_1$ with in-neighbors a_0, a_1 and $b \in T_2$ with in-neighbors b_0, b_1 we have*

$$\mathrm{match(a, b)} = 2^2 + \min_{\mathrm{i} \in \{0,1\}} \min_{\mathrm{j} \in \{0,1\}} (\mathrm{cost(T_1[a_i], T_2[b_j])} + \mathrm{cost(T_1[a_{1-i}], T_2[b_{1-j}])}).$$

Proof. Since $\delta(a) = \delta^-(b) = 2$, the minimum cost of a tree circuit T embedding both a and b is 2^2 plus the minimum cost of embedding the subtrees $T_1[a_0], T_1[a_1], T_2[b_0]$, and $T_2[b_1]$. We only need to check which of the four possible feasible combinations achieves the minimum. □

We now can finish the proof of Theorem 3 whose pseudo code is given as Algorithm 1.

Proof (Proof of Theorem 3). Consider Algorithm 1. We note that by proceeding in a reverse BFS-ordering of both $V(T_1)$ and $V(T_2)$ we ensure that we can compute $\mathrm{cost_1}$, $\mathrm{cost_2}$ and match in Lines 7,8 and 9. Hence, the correctness of this algorithms follows from Lemmas 3 and 4 and Corollary 1. Clearly the run time is equal to $O(|T_1||T_2|)$ times the runtime of determining $M[a_i, b_j]$ and $C[a_i, b_j]$. We consider these two parts separately. First, by Observations 4 and Lemma 4, determining $M[a_i, b_j]$ takes constant time. Hence, the total time taking determining the $|T_1| \times |T_2|$ array is $O(|T_1||T_2|)$. By Lemma 3, determining $C_1[a_i, b_j]$ takes $O(\delta^-(a) + 1)$ time. Hence, the total time determining C_1 is $\sum_{a_i} \sum_{b_j} O(\delta^-(a_i) + 1) = |T_2| \sum_{a_i} O(\delta^-(a_i) + 1) = O(|T_2||T_1|)$. Similarly, the total time determining C_2 is $O(|T_1||T_2|)$. The runtime now follow. Finally, determining an optimal tree is now trivial given the choices made by Algorithm 1. □

We finish this section by noting that to deal with XOR-gates, which are *free*, when two XOR gates are mapped to the same node in T, we ensure zero addition cost is added. With these modifications, it follows that Algorithm 1 can also be used to compute $\mathrm{cost(T_1, T_2)}$ in this more general case.

6.2 General Circuits

Heuristic Algorithm We develop a polynomial time heuristic algorithm using the machinery of Section 6.1. We then finish by sketching the proof of correctness. In

Input: Binary tree circuits T_1, T_2
Output: $\mathrm{cost}(\mathrm{T}_1, \mathrm{T}_2)$
let $a_1, ..., a_{n_1}$ be a BFS-ordering of $V(T_1)$
let $b_1, ..., b_{n_2}$ be a BFS-ordering of $V(T_2)$
for $i = n_1$ **down to** 1:
... **for** $j = n_2$ **down to** 1:
..... **determine** $M[a_i, b_j] = \mathrm{match}(\mathrm{a_i}, \mathrm{b_j})$.
..... **determine** $C_1[a_i, b_j] = \mathrm{cost}_1(\mathrm{T}_1[\mathrm{a_i}], \mathrm{T}_2[\mathrm{b_j}])$.
..... **determine** $C_2[a_i, b_j] = \mathrm{cost}_2(\mathrm{T}_1[\mathrm{a_i}], \mathrm{T}_2[\mathrm{b_j}])$.
..... **set** $C[a_i, b_j] = \min(C_1[a_i, b_j], C_2[a_i, b_j])$.
return $C(T_1[a_1], T_2[b_1])$

Algorithm 1: Determining $\mathrm{cost}(\mathrm{T}_1, \mathrm{T}_2)$

Section 8, we present the results of our experimental validation for this algorithm. For simplicity, assume every non-leaf node of T_1 and T_2 has weighted in-degree exactly two and we omit dealing with Free-XOR for now. We remark that again the ideas are easily extended to the general case.

We start by considering a related question. Let $D_1 = (V_1, E_1)$ and $D_2 = (V_2, E_2)$ be input circuit DAGs, each with exactly one output wire node. Let T_1 be a spanning in-arborescence subgraph of D_1 and let T_2 be a spanning in-arborescence subgraph of D_2. We determine a minimum cost circuit DAG D_0 embedding both D_1 and D_2 subject to the restriction that there must be a spanning in-arborescence subgraph T of D_0 such that (A) both T_1 and T_2 embed in T, and (B) leaves of T_1, resp. T_2, map to leaves of T. Denote the minimum cost of such a DAG by $\mathrm{cost}(\mathrm{D}_1|_{\mathrm{T}_1}, \mathrm{D}_2|_{\mathrm{T}_2})$ We remark that there always exists an appropriate choice of T_1 and T_2 such that D_0 will be a optimal embedding of D_1 and D_2. Further we remark, that we can essentially ignore Condition (B), since given any embedding of T_i, it is always possible to extend the out-arborescence of any leaf node of T_i down to a leaf node of T.

Analogous to Lemma 2, we can determine $\mathrm{cost}(\mathrm{D}_1[\mathrm{a}]|_{\mathrm{T}_1[\mathrm{a}]}, \mathrm{D}_2[\mathrm{b}]|_{\mathrm{T}_2[\mathrm{b}]})$ for $a \in T_1$ and $b \in T_2$ by considering $O(|T_1|+|T_2|)$ matchs.

Definition 6. Define the match* of $a \in D_1$ and $b \in D_2$ as the minimum cost of a circuit DAG D_0 such that there exists a mapping f_1 that embeds $D_1[a]$ into D_0 and a mapping f_2 that embeds $D_2[b]$ into D_0 such that $f_1(a) = f_2(b)$ and there exists a spanning in-arborescence subgraph T of D_0 such that (A) and (B) hold.

Definition 7. Let r be the root of circuit DAG D with gate nodes G. Further assume T is an in-arborescence subgraph of D containing r. Define the *cost of D on vertices of T* as $\mathrm{cost}_{\mathrm{T}}(\mathrm{D}) := \sum_{\mathrm{v} \in \mathrm{V}(\mathrm{T}) \cap \mathrm{G}} 2^{\mathrm{ffi}^-(\mathrm{v})}$.

Definition 8. Let T_1 and T_2 be circuit DAGs with roots a and b, respectively. Define:

(i) $\mathrm{cost}_2(\mathrm{D}_1|_{\mathrm{T}_1}, \mathrm{D}_2|_{\mathrm{T}_2}) := \min_{\mathrm{t} \in \mathrm{T}_2}(\mathrm{cost}_{\mathrm{T}_2}(\mathrm{D}_2) - \mathrm{cost}_{\mathrm{T}_2[\mathrm{t}]}(\mathrm{D}_2[\mathrm{t}]) + \mathrm{match}^*(\mathrm{a}, \mathrm{t}))$.

(ii) $\mathrm{cost}_1(\mathrm{D}_1|_{\mathrm{T}_1}, \mathrm{D}_2|_{\mathrm{T}_2}) := \min_{\mathrm{t}\in\mathrm{T}_1}(\mathrm{cost}_{\mathrm{T}_1}(\mathrm{D}_1) - \mathrm{cost}_{\mathrm{T}_1[\mathrm{t}]}(\mathrm{D}_1[\mathrm{t}]) + \mathrm{match}^*(\mathrm{t}, \mathrm{b}))$.

Lemma 5. *Let D_1 and D_2 be circuit DAGs. For $a \in D_1$ let T_1 be an in-arborescence subgraph of $D_1[a]$ containing a and for $b \in D_2$ let T_2 be a in-arborescence of $D_2[b]$ containing b. Then,*

$$\begin{aligned}&\mathrm{cost}(D_1[a]|_{T_1[a]}, D_2[b]|_{T_2[b]})\\ &\quad = \min\{\mathrm{cost}_1(\mathrm{D}_1|_{\mathrm{T}_1}, \mathrm{D}_2|_{\mathrm{T}_2}), \mathrm{cost}_2(\mathrm{D}_1|_{\mathrm{T}_1}, \mathrm{D}_2|_{\mathrm{T}_2})\}.\end{aligned}$$

From Lemma 5, in order to determine $\mathrm{cost}(\mathrm{D}_1[\mathrm{a}]|_{\mathrm{T}_1[\mathrm{a}]}, \mathrm{D}_2[\mathrm{b}]|_{\mathrm{T}_2[\mathrm{b}]})$, it remains to show how to determine $\mathrm{match}^*(\mathrm{a}, \mathrm{b})$. As before, if either $T_1[a]$ or $T_2[b]$ is a singleton then $\mathrm{match}^*(\mathrm{a}, \mathrm{b})$ is as follows.

Observation 5 *If $T_1[a]$ is a singleton, then for all $b \in T_2$, $\mathrm{match}^*(\mathrm{a}, \mathrm{b}) = \mathrm{cost}_{\mathrm{T}_2[\mathrm{b}]}(\mathrm{D}_2[\mathrm{b}])$. If $T_2[b]$ is a singleton, then for all $a \in T_1$, $\mathrm{match}^*(\mathrm{a}, \mathrm{b}) = \mathrm{cost}_{\mathrm{T}_1[\mathrm{a}]}(\mathrm{D}_1[\mathrm{a}])$.*

When neither $T_1[a]$ nor $T_2[b]$ is a singleton then whenever $\delta^-(a) = \delta^-(b) = 2$ we determine $\mathrm{match}^*(\mathrm{a}, \mathrm{b})$ as follows. For $a \in T_1$ with in-neighbors a_0, a_1 and $b \in T_2$ with in-neighbors b_0, b_1 we have $\mathrm{match}^*(\mathrm{a}, \mathrm{b})$ is equal to:

$$\begin{aligned}2^2 + \min_{i\in\{0,1\}} \min_{j\in\{0,1\}} &(\mathrm{cost}(\mathrm{D}_1[\mathrm{a_i}]|_{\mathrm{T}_1[\mathrm{a_i}]}, \mathrm{D}_2[\mathrm{b_j}]|_{\mathrm{T}_2[\mathrm{b_j}]})\\ &+ \mathrm{cost}(\mathrm{D}_1[\mathrm{a}_{1-\mathrm{i}}]|_{\mathrm{T}_1[\mathrm{a}_{1-\mathrm{i}}]}, \mathrm{D}_2[\mathrm{b}_{1-\mathrm{j}}]|_{\mathrm{T}_2[\mathrm{b}_{1-\mathrm{j}}]})).\end{aligned}$$

The case when the degrees do not match up is more complicated. Indeed, either the node a is incident to an edge which goes between two subtrees of $\mathcal{F}_1$ or the node b is incident to an edge which goes between two subtrees of $\mathcal{F}_2$. In this case the match* is undefined. To get beyond this, we consider two cases separately. First, if a is a leaf node in T_1 and b is a leaf node in T_2 then we need to create a dummy gate node which takes as input $f_1(a)$ and $f_2(b)$. Such a construction has match$^* = 12$ since we suffer cost 4 for each of $f_1(a)$, $f_2(b)$ and the dummy gate. Second, assume a is not a leaf node in T_1, the case when b is not a leaf in T_2 is symmetric. Our heuristic then sets match* equal to the minimum cost of a tree such that $f_1(a)$ to be the in-neighbor of $f_2(b)$.

We now can determine D_0 using the following variant of Algorithm 1.

Circuit DAG Embedding Algorithm

1. Chose a spanning in-arborescence forest $\mathcal{F}_1$ of D_1 such that one in-arborescence of $\mathcal{F}_1$ contains each output node of D_1. Similarly, choose $\mathcal{F}_2$ of D_2. In our implementation, we will focus on choosing such forests uniformly at random. Such forests can be found by choosing a single edge from each of the out-edges for each node of D_1 and D_2; we omit further details.
2. For each $T_1 \in \mathcal{F}_1$ and $T_2 \in \mathcal{F}_2$ compute $\mathrm{cost}(\mathrm{D}'_1|_{\mathrm{T}_1}, \mathrm{D}'_2|_{\mathrm{T}_2})$, where D'_1, respectively D'_2, is DAG found by taking the union of all edges of D_1, respectively D_2, with at least one end-point in T_1, respectively T_2. Here we can apply Algorithm 2 directly.

Input: D_1, D_2, T_1 and T_2
Output: $\mathrm{cost}(\mathrm{D}_1|_{\mathrm{T}_1}, \mathrm{D}_2|_{\mathrm{T}_2})$
let $a_1, ..., a_{n_1}$ be a BFS-ordering of $V(T_1)$
let $b_1, ..., b_{n_2}$ be a BFS-ordering of $V(T_2)$
for $i = n_1$ **down to** 1:
... **for** $j = n_2$ **down to** 1:
..... **determine** $M[a_i, b_j] = \mathrm{match}^*(\mathrm{a_i}, \mathrm{b_j})$.
..... **determine** $C[a_i, b_j] = \mathrm{cost}(\mathrm{D}_1[\mathrm{a_i}]|_{\mathrm{T}_1[\mathrm{a_i}]}, \mathrm{D}_2[\mathrm{b_j}]|_{\mathrm{T}_2[\mathrm{b_j}]})$.
return $C(T_1[a_1], T_2[b_1])$

Algorithm 2: Determining $\mathrm{cost}(\mathrm{D}_1|_{\mathrm{T}_1}, \mathrm{D}_2|_{\mathrm{T}_2})$

3. Using the costs computed in Step 2, we compute an optimal pair of in-arborescences as follows. Let G be the weighted bipartite graph with bipartition (A, B) defined as follows. Let $m := \max\{|\mathcal{F}_1|, |\mathcal{F}_2|\}$. The set A contains a node labeled by each in-arborescences of $\mathcal{F}_1$ plus $m - |\mathcal{F}_1|$ 'dummy' nodes. Similarly, the set B contains a node labeled by each in-arborescences of $\mathcal{F}_2$ plus $m - |\mathcal{F}_2|$ 'dummy' nodes. G is a complete bipartite graph, where an edge between a node of A labeled by T_1 and node B labeled by T_2 has weight $\mathrm{cost}(\mathrm{D}'_1|_{\mathrm{T}_1}, \mathrm{D}'_2|_{\mathrm{T}_2})$, and an edge between a dummy node and node of A labeled by T_1 has weight $\mathrm{cost}_{\mathrm{T}_1}(\mathrm{D}'_1)$, respectively node of B labeled by T_2 has weight $\mathrm{cost}_{\mathrm{T}_2}(\mathrm{D}'_2)$.
   Since G is complete, it has a perfect matching. Moreover, any perfect matching corresponds to a pairing of output nodes in D_1 with output nodes in D_2, where nodes matched to 'dummy' nodes have no partner and are embedded as a copy of themselves. Hence, it follows that a minimum cost matching in B corresponds to a minimum cost pairing of output nodes. We remark, that computing such a minimum cost perfect matching in time polynomial in $|A|+|B|$ is a classical result (see for e.g. [9]).
4. We now determine the final circuit DAG. For each $T_1^i - T_2^j$ pairing from the minimum cost perfect matching, we construct a tree circuit T^{i-j} and embeddings $f_1^{i-j} : T_1^i \to T^{i-j}$ and $f_2^{i-j} : T_2^j \to T^{i-j}$, where 'dummy' pairings are the identity embedding.
   Let $T = \bigcup_{i,j} T^{i-j}$. An embedding f_1 of $\mathcal{F}_1$ into T is found by taking the union of the f_1^{i-j} over all $T_1^i - T_2^j$ pairings (including 'dummy' nodes). An embedding f_2 of $\mathcal{F}_2$ into T is found in a similar way. Let D_0 be the DAG found by taking a copy of T. First, we add an edge from the source of $f_1(x)$ to the source of $f_1(y)$ of weight w'_{xy} for each edge $xy \in E_1 - E(\mathcal{F}_1)$. For each edge $xy \in E_2 - E(\mathcal{F}_2)$ we do the same though adding these edges might cause cycles. Before adding xy, we test if there exists a directed path from y to x in D_0. If such a path P exists, then there must exists an edge of P only used by the circuit D_1. By splitting the path up to this edge, we can insure that D_0 plus xy is acyclic. We then update f_1 and f_2 to include these additional edge mappings.

We complete the proof by showing that D_0 is a feasible solution.

Theorem 4. *The* Circuit DAG Embedding Algorithm *finds a feasible circuit DAG.*

Proof. Without loss of generality, it is enough to show that f_1 is a valid embedding of D_1 into D_0. By construction f_1 is a mapping from nodes of D_1 to out-arborescences of D_0 and from edges of D_1 to edges of D_0. We need only verify that Conditions 1, 2 and 3 of Definition 1 hold. Since Condition 1 holds for f_1^i and the perfect matching ensures that every vertex of D_1 is in exactly one paired embedding, Condition 1 holds for f_1. Conditions 2 and 3 hold, since either an edge is mapped by some f_1^i, satisfying Conditions 2 and 3, or the edge goes between trees of $\mathcal{F}_1$, where Step 4 adds these edges between sources of out-arborescences of weight satisfying Condition 3. It now follows that D_0 is a feasible solution. □

7 Optimally embedding graphs is NP-complete

Here we consider the complexity of the problem of finding a minimum sized container digraph D_0 embedding two digraphs D_1 and D_2. The problem is seen to be NP-complete via a reduction from 3-sat. The full details of the proof are deferred to the full version.

Theorem 5. *The problem of finding a digraph D_0 such that embedding two digraphs D_1 and D_2 into D_0 has cost at most k is NP-complete even when the in-degree and out-degree of each node in D_1 and D_2 is bounded by 2 and at most one of D_1 or D_2 is a tree.*

8 Experimental Evaluation and Validation

Here we report on our experimental evaluation of the heuristic given in Section 6, as well as on the resulting efficiency of Protocol 1 in comparison with standard GC and Protocol 2 in comparison with standard GMW.

Evaluation methodology. The main metric we use to compare our approach to GC is the *total* bandwidth required, consumed by *all* OT instances and garbled gates transfers. We do not penalize ourselves for the potential increase in latency due to additional round per `switch`, associated with our approach. As discussed in the Introduction, this is because in large circuit/batch execution round trip delays may overlap with data transmission, and, if so, latency will not impact performance. Of course, in some scenarios (e.g. large network latency, small circuit/single execution) latency may dominate. We leave full implementation and parameters tuning as important future work to address these settings.

We stress that for the SPF-SFE and GMW case, where we report significant concrete improvement in our experiments, we do not require additional rounds as compared to standard GC.

We validate our approach with the experiments on a set of circuits which we built using circuit compiler CBMC-GC [8], summarized in Table 1. We constructed these 32 circuits by exploiting variations and combinations of a number of available arithmetic and bit-operation circuits. Because of this, there are commonalities among the input circuits/DAGs (which is typical in practice), and which may be affecting performance. We stress that our algorithms are not aware of the commonalities in the circuits and apply generically. We further note that these circuits are not hand-optimized for the functions they compute. Indeed, our goal is not to find the best circuit for a specific function, but to validate our heuristic and to understand its behavior. We do this by running it on a set of simple circuits of varying sizes and similarity for our experiments. In many applications (e.g., private DB policies) the clauses would be more similar, and we expect even better performance.

Results. Firstly, we stress that our heuristics are still *highly unoptimized*. Even with this, we are able to determine container circuit C_0 containing all 32 input circuits $C_1, ..., C_{32}$ whose size is 0.1637 times the size of all circuits taken together. To explain further, we note that the size of C_0 is trivially at least $\max_{i=1..32}\{|C_i|\}$ and at most $\sum_{i=1}^{32}|C_i|$. Here $|C_i|$ denotes the cost of a circuit including free-XOR[6]. As we will explain, the size of C_0 compared to these bounds yields an important metric for the performance of the algorithm. Formally, we define the *expansion metric, or EM* as $m = \frac{|C_0| - \max_{i=1..32}\{|C_i|\}}{\sum_{i=1}^{32}|C_i|}$[7]. Clearly, $m \in [0, 1]$ where values closer to 0 indicate better performance of the algorithm.

Starting with the 32 input circuits, we first heuristically determine over 100 random trials the smallest circuit containing each of the $\binom{32}{2}$ pairs. For a particular pair of circuits C_i, C_j, we define the *round EM* to be the minimum over all random trials of $\frac{|C_0| - \max\{|C_i|,|C_j|\}}{|C_i|+|C_j|}$. Given all these container circuits, we choose the pairing of circuits of minimum total size. We use these 16 resultant circuits as the input circuits for the next round and repeat the process.

Table 2 compares the total number of non-free gates for a $\mathcal{S}$-Universal Circuit, $\mathcal{S} = \{C_1, ..., C_{32}\}$, using existing approaches and our work. In Figure 4, we report the total size of circuits in each of the five rounds, resulting in total size reduction of 6.1×.

In the full version, we include additional discussion on the experiment results and anticipated behavior.

[6] We remark that in all our experiments we use circuits that have fan in at most 2. A standard reduction allows us to eliminate gates of fan in exactly 1. In our experiments we use cost and size interchangeably, since they are closely related.

[7] It would be more general to include the size of Valiant's universal circuit in the expansion metric definition. For s defined as the size of a universal circuit for all circuits of size up to $\max\{|C_i|\}$, set EM $m = \frac{|C_0| - \max\{|C_1|,...,|C_{32}|\}}{\min\{s,|C_1|+..+|C_{32}|\}}$. However, in our experiments and clause numbers, s is much larger than $|C_1|+.. + |C_{32}|$, so we omit this complication in this writeup.

Circuit Test #	Function	Total # of Gates	# XOR Gates
C_1	$A + B$ (32bit)	154	123
C_{17}	$A + B$ (16bit)	74	59
C_{16}	$A - B$ (16bit)	103	74
C_{32}	$A - B$ (32bit)	215	154
C_{18}	$A < B$ (32bit)	191	127
C_{19}	$A < B$ (16bit)	74	63
C_2	$A \leq B$ (32bit)	191	127
C_3	$A \leq B$ (16bit)	95	63
C_4	Hamming (32 bit)	1610	1223
C_{20}	Hamming (16bit)	775	587
C_5	Integer Division (32bit)	3283	1925
C_{21}	Integer Division (16bit)	3225	1830
C_7	$A * B$ (32bit)	3283	1925
C_{22}	counting loop (16bit)	1490	494
C_{25}	$A + B < 230$ and $A - B > 20$ (32bit)	487	333
C_9	$A + B < 230$ and $A - B > 20$ (16bit)	231	157
C_{27}	$A * B > 200$ (32bit)	3368	1982
C_{11}	$A * B > 200$ (16 bit)	904	478
C_{23}	$A * B$ (16bit)	867	453
C_{24}	$A + B < 100$ (32bit)	183	122
C_8	$A + B < 100$ (16bit)	87	58
C_{26}	$B > 1020$ and $A * B > 10$ (32bit)	3458	2037
C_{10}	$B > 1020$ and $A * B > 10$ (16bit)	946	501
C_{28}	$A * B > B + 10 * A$ (32bit)	3881	2311
C_{12}	$A * B > B + 10 * A$ (16bit)	1145	631
C_{29}	$B * A + 555$ (32bit)	3343	1956
C_{13}	$B * A + 555$ (16bit)	895	468
C_{30}	$B^2 + A^2 > 1$ (32bit)	5613	3881
C_{14}	$B^2 + A^2 > 1$ (16bit)	1373	905
C_{31}	$B^2 + A * B + A^2$ (32bit)	8660	5809
C_{15}	$B^2 + A * B + A^2$ (16bit)	2132	1361
C_6	leading bit (16bit)	221	74

Table 1. Circuits used for heuristic evaluation.

Table 2. Total non-free gate counts for a $\mathcal{S}$-Universal Circuit, $\mathcal{S} = \{C_1, ..., C_{32}\}$.

Combined Circuit	Valiant Universal Circuit	Our construction
20,543	562,900	3,363

Acknowledgment. The second author was supported by the Office of Naval Research (ONR) contract number N00014-14-C-0113.

References

1. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 535–548. ACM Press, November 2013.
2. Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.
3. Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In Reihaneh Safavi-Naini and Jennifer Seberry, editors, *ACISP 03*, volume 2727 of *LNCS*, pages 350–364. Springer, Heidelberg, July 2003.
4. Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001.

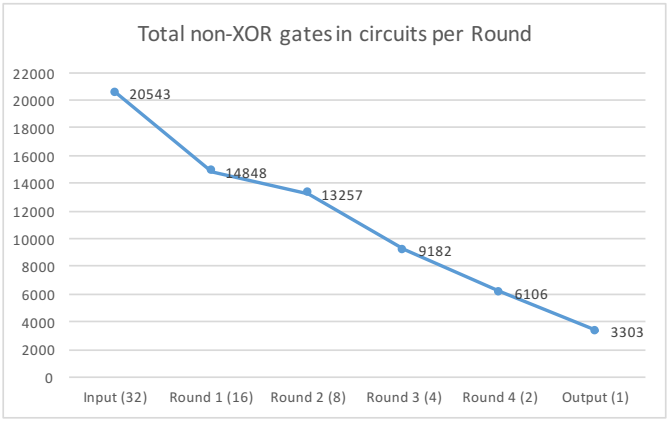


Fig. 4. Determining a container circuit C_0 containing all 32 input circuits $C_1, ..., C_{32}$. Reported is the total non-XOR gates in the intermediate container circuits. Figure 5 reports the actual pairings of circuits in each round.

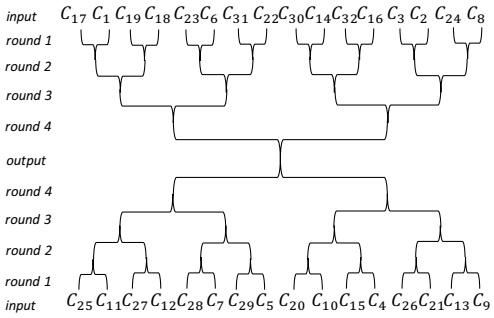


Fig. 5. The circuit pairing found.

5. Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 1504–1517. ACM Press, October 2015.
6. Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *24. Annual Network and Distributed System Security Symposium (NDSS'17)*. The Internet Society, February 26-March 1, 2017. To appear.
7. Ben A. Fisch, Binh Vo, Fernando Krell, Abishek Kumarasubramanian, Vladimir Kolesnikov, Tal Malkin, and Steven M. Bellovin. Malicious-client security in blind seer: A scalable private DBMS. In *2015 IEEE Symposium on Security and Privacy*, pages 395–410. IEEE Computer Society Press, May 2015.

8. Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In Albert Cohen, editor, *Compiler Construction - 23rd International Conference, CC 2014*, volume 8409 of *LNCS*, pages 244–249. Springer, 2014.
9. Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, CA, 1979.
11. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge University Press, 2009.
12. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
13. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.
14. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.
15. Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 699–728. Springer, Heidelberg, May 2016.
16. Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 54–70. Springer, Heidelberg, August 2013.
17. Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, Heidelberg, August 2014.
18. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.
19. Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In Gene Tsudik, editor, *FC 2008*, volume 5143 of *LNCS*, pages 83–97. Springer, Heidelberg, January 2008.
20. Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.
21. Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao, editors, *ISC 2005*, volume 3650 of *LNCS*, pages 314–328. Springer, Heidelberg, September 2005.
22. Helger Lipmaa, Payman Mohassel, and Saeed Sadeghian. Valiant's universal circuit: Improvements, implementation, and applications. Cryptology ePrint Archive, Report 2016/017, 2016. `http://eprint.iacr.org/2016/017`.
23. Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 557–574. Springer, Heidelberg, May 2013.
24. Rafail Ostrovsky and William E. Skeith III. A survey of single-database private information retrieval: Techniques and applications (invited talk). In Tatsuaki Oka-

moto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 393–411. Springer, Heidelberg, April 2007.
25. Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE Computer Society Press, May 2014.
26. Annika Paus, Ahmad-Reza Sadeghi, and Thomas Schneider. Practical secure evaluation of semi-private functions. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *ACNS 09*, volume 5536 of *LNCS*, pages 89–106. Springer, Heidelberg, June 2009.
27. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.
28. Neil Robertson and P.D. Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39 – 61, 1983.
29. Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.
30. Leslie G. Valiant. Universal circuits (preliminary report). In *STOC*, pages 196–203, New York, NY, USA, 1976. ACM Press.
31. R. M. Verma and S. W. Reyner. An analysis of a good algorithm for the subtree problem, correlated. *SIAM J. Comput.*, 18(5):906–908, 1989.
32. Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of MIPS machine code. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 99–117. Springer, Heidelberg, September 2016.
33. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.
34. Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.