Striving for Both Quality and Speed: Logic Synthesis for Practical Garbled Circuits

Mingfei Yu, Giovanni De Micheli
Integrated Systems Laboratory, EPFL
Lausanne, Switzerland
{mingfei.yu, giovanni.demicheli}@epfl.ch

***Abstract*—Garbled circuit (GC) is one of the few promising protocols to realize general-purpose secure computation. The target computation is represented by a Boolean circuit that is subsequently transformed into a network of encrypted tables for execution. The need of distributing GCs among parties however requires excessive data communication, called garbling cost, which bottlenecks system performance. Due to the zero garbling cost of XOR operations, existing works reduce garbling cost by representing the target computation as the XOR-AND graph (XAG) with minimum multiplicative complexity. Recently, an XOR-OneHot graph (X1G) has been proposed as an efficient GC representation. However, there is a lack of formal proof of X1G performance in the literature. In this paper, we prove that starting from any XAG, there exists an X1G implementation with equal or lower garbling cost. Based on our findings, we propose (a) an affine function classification-based database generation method, which decouples time-consuming on-the-fly exact synthesis from Boolean rewriting; (b) a novel optimal X1G synthesis approach to accelerate the database generation procedure. The proposals jointly facilitate a performant Boolean rewriting-based X1G optimization method. Experimental evaluations show significant improvement in both garbling cost and runtime: with a 2273.63× speed-up achieved on average, the proposed method realized an up-to 8.20% improvement in garbling cost reduction compared to the state-of-the-art.**

***Index Terms*—logic synthesis, garbled circuits, secure multi-party computation, multiplicative complexity**

I. Introduction

Secure computation, also known as *secure multi-party computation* (MPC), refers to the process of computing a joint function on private inputs from multiple parties in a way that ensures the privacy and security of the inputs. This feature enables parties to collaborate in a privacy-preserving way and provides an ideal solution to many scenarios where privacy and security are paramount, such as financial transactions [1], voting systems [2], and medical data sharing [3].

However, realizing general-purpose MPC is not easy. Among the few candidates in the literature, *garbled circuit* (GC) is a promising one. First proposed by Andrew Yao in 1986 [4], GC is a cryptographic technique with a constant round complexity. GC offers several advantages over other competitors: compared to *fully homomorphic encryption* (FHE) [5], GC is free from a prohibitive computational overhead and can be executed much faster; in comparison with *secret sharing* [6], GC protects the privacy of intermediate values, preventing information about the private inputs from being revealed by intermediate results.

Meanwhile, this powerful technique has its limitations. In this protocol, the target computation is represented by a Boolean circuit and subsequently transformed into a garbled circuit — essentially a network of encrypted tables — for execution. The requirement that the GC shall be transmitted among parties results in an excessive amount of data communication. This communication cost, also known as *garbling cost*, is a significant factor that bottlenecks the overall system performance: since the size of a GC depends on the number of logic gates in the Boolean circuit, while modern application scenarios are typically large-scale, and commonly have requirements for low latency and high energy efficiency, garbling cost becomes a major concern.

To make the protocol more efficient, researchers have proposed various solutions, which generally fall into two categories. One focuses on improving the garbling scheme, because a smarter way to garble each logic gate can reduce the number of cipher-texts in each encrypted table. To name but a few: The *free-XOR* technique points out that, by making use of the algebraic property of XOR operations, garbling XOR gates can be free from resulting in any encrypted tables [7]. *Half-gate* shows that any 2-input non-linear logic operation, such as AND2, can be garbled using 2 cipher-texts, i.e., a 2-entry encrypted table [8]. *Garbling gadget* proposes that the garbling cost of any m-input symmetric logic gate is no more than m, suggesting that large-fanin-size logic gates might be more garbling cost-efficient non-linearity providers than AND2 [9].

The other technical direction, to which this work belongs, is to optimally synthesize the Boolean circuit that implements the target computation. Due to the fact that XOR gates and inverters are free of garbling cost (by adopting the free-XOR scheme), it is widely regarded as a *multiplicative complexity* (MC) *reduction* problem: implement the target Boolean function over the basis of {AND2, XOR2, NOT}, i.e., as an *XOR-AND graph* (XAG), then minimize the number of AND2s in the XAG through manipulating logic. Remarkable progress has been achieved by either exploiting existing logic optimization techniques with the number of AND2s targeted as the quality measure [10], or tailored approaches, such as detecting those AND2s that can be replaced by XNOR2s without changing the functionality of the circuit [11]. It is also of interest to find more garbling cost-efficient logic primitives to represent the Boolean circuits to be garbled: inspired by garbling gadget, a study on the garbling cost-efficiency of larger-fanin-size logic gates reveals that both wide AND and OneHot3 are more efficient non-linearity providers than AND2; thus, it is preferable to adopt compact XAGs (generalized XAGs that support wide ANDs) or XOR-OneHot graphs (X1Gs) over XAGs as the logic representation when synthesizing the Boolean circuits [12]. However, there is a lack of formal comparison between the performance of these two garbling cost-efficient logic representations, as well as supportive logic optimization techniques.

For the first time, we report in Section III that X1G is the better logic representation for the task of practical GC generation, evidenced by the observation that starting from any compact XAG, there always exists an X1G implementation with equal or lower garbling cost. To facilitate a powerful and agile X1G-oriented logic optimization method, two proposals are made, respectively in Section IV and V, to efficiently develop a database of garbling cost-optimal X1G implementations for 6-variable Boolean functions: (a) applying affine function classification to limit the database to a manageable size; and (b) a novelly-formulated exact synthesis approach that can quickly synthesize optimal X1G for small-scale Boolean functions. As reported in Section VI, evaluations on three popular benchmark suites show that our method on average outperforms the state-of-the-art algorithm in garbling cost reduction by respectively 1.97%, 1.59%, and 0.51%, together with 2441.64×, 795.58×, and 3950.08× speed-ups.

II. Background

A. Garbled Circuits Protocol

The concept of GC was first proposed by Yao as a solution to secure two-party computation (2PC) [4] and is later generalized to an MPC protocol [13]. We introduce its 2PC version for clarity, which is also the core of the MPC version. Generally, we adopt the formalization provided in [14].

Two parties, namely Alice and Bob, would like to rely on GC to collaborate on computing a function f without revealing their private inputs to each other. They are supposed to respectively play the roles of *garbler*, who is in charge of generating the GC, and *evaluator*, who evaluates the generated GC. Without loss of generality, we assume that Alice is the garbler and Bob is the evaluator.

The execution of the protocol can be split into 5 steps:

1) *Boolean circuit generation*. Starting from a target computation, commonly described in high-level languages like C++ or Python, it is required that Alice generates a Boolean circuit whose function $f(\cdot)$: $\mathbb{B}^m \rightarrow \mathbb{B}^n$ implements the computation.
2) *GC generation*. With a parameter $l \in \mathbb{N}$ indicating the desired security level, for each wire in the circuit, Alice selects an encoding function $\mathit{En}(\cdot)$: $\mathbb{B} \rightarrow \mathbb{B}^l$ that maps the two potential binary value, 0 and 1, to two l-bit labels. For example, the label A that corresponds to Alice's private input a is created following $A = \mathit{En}(a)$. Correspondingly, for each logic gate in the circuit, an encrypted table is created based on its truth table; each entry of the table is a cipher-text created by encrypting the output label using the input labels. In this way, a GC F, which is essentially a network of encrypted tables, is generated by Alice on top of the Boolean circuit.
3) *GC transmission*. Alice sends Bob: the GC F; the label corresponding to her inputs A; and a set of labels $\mathcal{B}$ that consists of all the potential labels for Bob's private input. By exploiting *oblivious transfer* (OT) as the moderator, among the labels in $\mathcal{B}$, Bob only learns B, the one that corresponds to his private input b.
4) *GC evaluation*. Bob evaluates the received GC F and obtains the garbled output Y following $Y = F(X)$, where $X = \{A, B\}$.
5) *Result sharing*. Alice announces the decoding function for the primary output wire $\mathit{De}(\cdot)$: $\mathbb{B}^l \rightarrow \mathbb{B}$, while Bob shares the evaluation result Y. These two resources of information jointly determine the computation result y, as $y = \mathit{De}(Y)$.

The need of sending the generated GC from Alice to Bob in the GC transmission stage typically results in an excessive amount of data communication.

B. Role of Multiplicative Complexity in Practical GC

Free-XOR [7] connects synthesizing practical GC to a well-known logic synthesis problem, the MC reduction problem.

Given a Boolean function, its MC is the minimum number of AND2s sufficient to implement it exclusively using AND2s, XOR2s, and inverters (i.e., as an XAG) [15]. However, figuring out the MC of a Boolean function is generally an intractable problem [16]. Thus, depending on the context, MC also commonly refers to the AND2 count in an XAG implementation of a Boolean function.

To avoid ambiguity, *functional MC* and *structural MC* are used to distinguish the two cases [12]: for a given function, its functional MC is one of its attributes, while structural MC is a feature of an XAG implementation of it. Indeed, a Boolean function's functional MC lower-bounds the structural MC of any XAG implementing this function.

The MC reduction problem asks: "Starting from an XAG, how to reduce the number of AND2s in it as much as possible by manipulating logic synthesis techniques?" Since among the logic primitives of {AND2, XOR2, NOT}, AND2 is the only one that introduces garbling cost to the synthesized GC, any progress in addressing the MC reduction problem directly contributes to the synthesis of lower-cost GCs.

C. Garbling Cost-Efficiency of Symmetric Logic Gates

It was first proposed in [9] that symmetric logic gates, whose output depends on the Hamming Weight of the input pattern, can be garbled more efficiently than believed.

This is realized by interpreting a symmetric logic operation as a modular addition followed by an integer-to-binary projection. For an m-input symmetric logic operation, such a modulus n is no more than $m + 1$. Since a modular addition is free of garbling cost, the garbling cost of an m-input symmetric logic gate is numerically the garbling cost of an n-to-2 projection, which is upper-bounded by $n-1=m$ [17].

Due to the unateness of AND operation, the modulus required to interpret an m-input AND operation is always $m+1$, determining that garbling an m-input AND gates results in m cipher-texts. On the other hand, since each m-input AND decomposes into $m-1$ AND2s, the functional MC of an m-input AND gate is $m-1$.

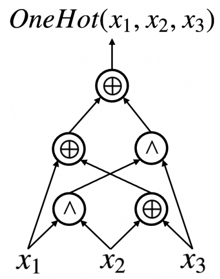


Fig. 1: Structual MC-Optimal XAG that Implements OneHot3

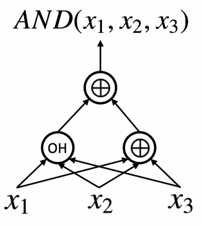


Fig. 2: Implement AND3 using OneHot3s and XORs

Meanwhile, a OneHot3 operation, whose truth table is $\#16$[1], can be garbled using only 2 cipher-texts due to its property that its outputs are the same when given the input patterns with the minimum or maximum Hamming Weight (i.e., 000 and 111). The functional MC of the OneHot3 operation is 2, as at least two AND2s are required to implement it as an XAG (Fig. 1, "$\wedge$" and "$\oplus$" denote AND and XOR respectively).

Non-linear logic gates are indispensable to represent a computation as a Boolean circuit. It is impossible to build any Boolean function using merely XORs and inverters, as witnessed by the functional MC of the function. A logic gate's garbling cost-efficiency in providing structural MC is defined as its *MC compactness* [12]. When adopting AND2 as the logic primitive to provide non-linearity, at least 2 cipher-texts are required to introduce one unit of structural MC to the resulting logic network [8]. In contrast, an m-input AND gate offers $m-1$ units of structural MC at the expense of m cipher-texts, while a OneHot3 gate spends 2 cipher-texts to introduce 2 units of structural MC. Hence, both wide AND and OneHot3 are more garbling cost-efficient non-linearity providers compared to AND2.

For this reason, regarding garbling cost reduction as an MC reduction problem has its limitations. This formulation depends on the premise that XAG is adopted as the logic representation for the Boolean circuit, which is no longer plausible: by either (a) generalizing XAG to compact XAG, where the fanin size of ANDs is allowed to be arbitrary, or (b) adopting X1G instead as the logic representation, a reduced garbling cost can be achieved.

D. Affine Function Classification

Affine operation refers to those transformations that, if applied, do not change the algebraic properties of a Boolean function [18]. Based on an n-variable Boolean function $f(x_1, \cdots, x_i, \cdots, x_j, \cdots, x_n)$, the five affine operations are:

1) Swap two variables: $f \xrightarrow{x_i \leftrightarrow x_j} f'$.
2) Complement a variable: $f \xrightarrow{x_i \rightarrow \neg x_i} f'$, where "$\neg$" indicates negation.
3) Complement the function: $f \xrightarrow{\neg} f'$.
4) Translational operation: $f \xrightarrow{x_i \rightarrow (x_i \oplus x_j)} f'$.
5) Disjoint translational operation: $f \xrightarrow{\oplus x_i} f'$.

Two Boolean functions, f and g, are defined as *affine equivalent* if there is a series of affine operations $\mathbf{o} = \{o_1, \cdots, o_k\}$ that satisfies

$$f \xrightarrow{o_1} \cdots \xrightarrow{o_k} g.$$

In some applications, such as cryptography, *affine equivalence* theory provides a powerful approach to classify Boolean functions, which we hereafter refer to as *affine function classification*. For example, all 2^{16} 4-variable Boolean functions split into 222 NPN equivalence classes [19] but only 8 affine equivalence classes [20].

III. Wide AND or OneHot

Compared to AND2, both large-fanin-size AND and OneHot3 are more garbling cost-efficient MC providers. However, we have noticed that given a Boolean function, any of its compact XAG implementations can be converted into an X1G with equal or lower garbling cost, implying that X1G is the better logic representation for low-cost GC generation.

A. One-to-One Conversion from a Compact XAG to an X1G

By adopting garbling gadget as the garbling scheme, garbling an AND3 gate and a OneHot3 gate requires only 3 and 2 cipher-texts respectively [9]. This points out the possibility of making use of OneHots to garble AND operations with a reduced cost. As can be learned from the *algebraic normal form* (ANF) [16] of the OneHot3 operation: $OneHot3(x_1, x_2, x_3) = x_1x_2x_3 \oplus x_1 \oplus x_2 \oplus x_3$, we can extract the two inherent AND2s in it by applying extra XORs to cancel the last three items in the formula. Such a solution is depicted in Fig. 2, where an AND3 operation is realized using one OneHot3 (we use "OH" to denote a OneHot gate) and two XORs. Thanks to the fact that garbling XORs is free of cipher-texts, the implementation requires 2 cipher-texts in total, cheaper than directly garbling an AND3 operation.

It is believed that adopting OneHots as the logic primitives to introduce structural MC is effective in reducing garbling cost but needs considerable runtime. This is because the starting point of the logic synthesis flow is assumed to be an optimized XAG (in order to take advantage of existing achievements on MC reduction); a time-consuming logic network restructuring is therefore inevitable to obtain an X1G from an XAG. On the other hand, the adoption of wide AND is relatively easy, as an XAG can be quickly transformed into a compact XAG by merging adjacent AND2s [12]. Our aforementioned observation however suggests that a compact

[1]In this paper, we represent truth tables in hexadecimal as a bit-string and the most significant bit is on the left-hand side.

Algorithm 1: Converting a compact XAG into an X1G with equal or lower garbling cost

```
Input: A compact XAG, $N$
Output: An functionally-equivalent X1G
1  foreach AND node $n \in N$ do
2      $\mathcal{M} \leftarrow$ decompose $n$ into consecutive AND2s
3      while $\mathcal{M} \neq \emptyset$ do
4          if $|\mathcal{M}| = 1$ then
5              $\{x_1,x_2\}\leftarrow$ fanins of $\mathcal{M}[0]$
6              n'$\leftarrow OneHot(1, \neg x_1, \neg x_2)$
7              replace node $\mathcal{M}[0]$ with n'
8              remove $\mathcal{M}[0]$ from $\mathcal{M}$
9          else
10             $\{x_1,x_2\}\leftarrow$fanins of $\mathcal{M}[0]$
11             $x_3 \leftarrow$the non-$\mathcal{M}[0]$ fanin of $\mathcal{M}[1]$
12             n'$\leftarrow$implement $AND(x_1, x_2, x_3)$ as Fig. 2
13             replace nodes $\mathcal{M}[0]$ and $\mathcal{M}[1]$ with n'
14             remove $\mathcal{M}[0]$ and $\mathcal{M}[1]$ from $\mathcal{M}$
15 return $N$
```

XAG can be easily converted into an X1G, whose garbling cost is provably never higher than before conversion.

Algo. 1 has a linear time complexity. It takes a compact XAG and goes through all the AND gates in it. When an m-input AND is encountered, it would be regarded as $m-1$ consecutive AND2s (line 2). Each pair of AND2s is garbled following the solution shown in Fig. 2 (line 10-13); If there is an AND2 that cannot form a pair, which happens when m is even, by following the rule that

$$AND(x_1, x_2) = OneHot(1, \neg x_1, \neg x_2),$$

this remaining AND2 is replaced by a OneHot gate (line 5-7).

Proposition 1. *The garbling cost of the X1G obtained by applying Algo. 1 is never higher than the cost of the starting point compact XAG.*

Proof. Assume there are p ANDs in the original compact XAG, whose fanin sizes are respectively m_1, m_2, $\cdots$, m_p. Recall that garbling an m-input AND operation requires m cipher-texts, then the garbling cost of this compact XAG is

$$cost_{XAG} = \sum_{i=1}^{p} m_i. \tag{1}$$

In contrast, by converting this compact XAG into an X1G via applying Algo. 1, the resulted garbling cost becomes

$$cost_{X1G} = \sum_{i=1}^{p} (2 \cdot \left\lceil \frac{m_i - 1}{2} \right\rceil), \tag{2}$$

which is in the range of

$$\sum_{i=1}^{p} m_i - p \leq cost_{X1G} \leq \sum_{i=1}^{p} m_i.$$

Notice that $cost_{XAG}$ is numerically equal to the upper bound, which would be reached only if all m_i-s are even. QED

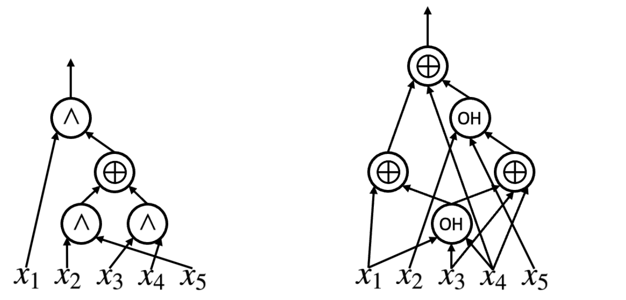
We further provide experimental evidence by generating the X1G implementations for the 10 benchmarks in the EPFL benchmark suite of random/control circuits. We adopted the merging algorithm in [12] to generate the compact XAGs to feed to Algo. 1. It averagely spends merely 0.02s on a laptop computer to realize a 4.20% garbling cost reduction.

B. Limitations of Relying on One-to-One Conversion

Algo. 1 not only suggests that X1G is the better choice of logic representation for practical GC generation but also enlightens us with a runtime-friendly way to benefit from its garbling efficiency. However, it is further noticed that relying on this one-to-one conversion from compact XAG to X1G (Algo. 1) can direct us to sub-optimality.

Eq. 1 and 2 respectively define the garbling costs of a compact XAG and an X1G: a compact XAG implementation is preferable if it consists of fewer but larger-fanin-size ANDs; while a better X1G implementation is the one with fewer One-Hot3s in it. Thus, when garbling cost-optimally implementing a function, its compact XAG-based and X1G-based solutions are likely different in structure.

We take as an example the task of finding the garbling cost-optimal implementation for a function whose truth table is $^{\#}2888a000$. This function is one of the representative functions of all 48 affine equivalence classes for 5-variable Boolean functions and has a known functional MC of 3.



(a) Compact XAG-Based (b) X1G-Based

Fig. 3: Garbling Cost-Optimal Implementations of $^{\#}2888a000$

Naïvely applying Algo. 1 to the compact XAG in Fig. 3a cannot lead us to the X1G in Fig. 3b — such an application would result in an X1G with 3 OneHot3s, whose garbling cost is 6, which is 50% worse than the optimal. Indeed, the expected compact XAG that can be converted into the optimal X1G shall be the one with two AND3s in it. However, the structural MC of such a compact XAG is 4, while the structural MC of the compact XAG in Fig. 3a is only 3, which is numerically equal to the functional MC of the target function and is therefore minimal. That being said, given a target function, its garbling cost-optimal X1G is not always reachable if we rely on applying the one-to-one conversion to its garbling cost-optimal compact XAG implementation.

The two observations proposed in this section jointly suggest that our goal of generating practical GCs can be simplified as: given a target function, figuring out a way to efficiently find its garbling cost-optimal X1G implementation.

IV. Affine Function Classification-Based Database Generation

In the previous section, we found the optimal implementations for the exemplary function via *exact synthesis*. It is a logic synthesis technique that involves finding the most efficient way to represent Boolean functions using network primitives but typically suffers from poor scalability [21]. In fact, most logic synthesis techniques are heuristic, as the sizes of functions encountered in practical problems are too large, making the pursuit of global optimum infeasible.

A large portion of successful heuristics can be categorized as *peephole optimizations* [22]. As its name indicates, peephole optimization refers to those divide-and-conquer-based strategies, which focus and optimize a partial logic network at a time. Existing effort on X1G-oriented logic optimization falls precisely into this category: in [12], the authors propose a Boolean rewriting-based X1G optimization method, where an exact synthesis solver is invoked on-the-fly to find the optimal X1G implementation for each encountered partial network. Due to the intrinsic complexity of exact synthesis, this approach requires a prohibitive runtime. The demand for low latency in most application scenarios of MPC determines the desire for a runtime-friendly X1G optimization technique.

A database of optimal X1G implementations for small-scale Boolean functions can effectively eliminate the need of invoking exact synthesis in an on-the-fly manner and significantly reduce the runtime of Boolean rewriting. Therefore, in this section, we aim at generating such a database.

A potential solution in the literature is *Boolean mining*-based functionally-incomplete database generation [23]. Unfortunately, it is inapplicable to our problem: Boolean mining requires that the Boolean functions that occur in practice are collected before entering the logic optimization procedure so that their optimal implementations can be prepared in advance; Considering the privacy-preserving property of MPC, such a premise is likely invalid. Hence, the database to be created should possess the characteristics of: (1) functionally complete and (2) with a reasonable number of entries.

Affine function classification turns out to be a natural fit. As introduced in Section II, for any two affine-equivalent Boolean functions f and g, there exists a series of affine operations $\mathbf{o}$ that converts f to g; The reverse process of $\mathbf{o}$, which converts g to f, is denoted by $\mathbf{o}'$. The reasonability of generating the database based on affine function classification is evidenced by the following proposition:

Proposition 2. *Assume the garbling cost-optimal X1G implementation of the function f is available, denoted as f_1. Then, by applying $\mathbf{o}$ to f_1, the obtained X1G g_1 is a garbling cost-optimal X1G implementation of the function g.*

Proof. If g_1 is not a garbling cost-optimal X1G implementation of the function g, there exists an X1G g_2 that also implements function g but requires fewer OneHot3s than g_1. By applying $\mathbf{o}'$ to g_2, another implementation of function f, f_2, can be obtained. Since applying affine operations never change the number of OneHot3s in a logic network, there are fewer OneHot3s in f_2 than in f_1, indicating that f_1 is not a garbling cost-optimal X1G implementation of the function f, which leads to a contradiction. QED

This is not the first attempt in the literature to adopt affine function classification in a logic synthesis problem. Existing trials can be found in tasks like MC reduction [24], synthesizing optimal quantum circuits [25], etc. But our proposal distinguishes itself as it is targeting X1G, while all the above-mentioned works target XAG as the objective logic representation.

Boolean rewriting relies on k-feasible *cut enumeration* [26] as a pre-process, to decide which partial networks the optimization process should focus on. The selection of k determines the allowed number of inputs to the partial networks. A larger k allows the process to get closer to the global optimum. Certainly, due to scalability concerns, it is impossible to set k to be unreasonably large. Configuring k to 4 or 5 is common in practice.

Thanks to the adoption of affine function classification, there are only 150 357 equivalence classes for all 6-variable Boolean functions, implying the feasibility of creating a database to support setting k as 6. A representative function can be assigned for each class [27]. However, finding the garbling-cost optimal X1G implementations for these 150 357 functions is still a non-trivial task. Thus, an approach to efficiently synthesizing optimal X1Gs for small-scale functions can significantly facilitate the generation of the targeted database.

V. A Novel Approach to Efficiently Synthesize Garbling Cost-Optimal X1Gs

A. Abstract XAG and AND Fence

In [16], the authors simplify the network description from the original XAG to a more general form. This representation, which is later termed as *abstract XAG* [28], features that: (1) fanin sizes of XORs are arbitrary (called *XOR clouds*); (2) each fanin of any XOR is either a primary input or an AND2 belonging to a lower logical level; (3) fanins of any AND2 are XOR clouds; (4) primary outputs are XOR clouds.

The number of steps of an abstract XAG equals the number of AND2s in it, i.e., between every two AND2s in adjacent steps, there are two XOR clouds inserted. It is observed in [28] that, by solving the SAT-based exact synthesis problem formulated on top of abstract XAG, the structural MC-optimal (i.e., AND2 count-minimal) implementation can be found significantly faster. We suppose the reason is that this encoding relaxes the restriction on the optimality in XOR count, which considerably increases the number of solutions in the search space and makes finding a solution much easier.

The XOR count in a solution found in this way is commonly beyond the minimal [28]. But considering that XORs are literally free in the context of GC generation, this complete lack of concern over XOR count makes this formulation a natural fit for our problem. To adapt this technique, we generalize it by supporting multi-fanin ANDs. To avoid ambiguity, we hereinafter use refer to our extended version as *abstract XAG*.

Additionally, in order to quickly evaluate the quality of an abstract XAG implementation, we introduce a concept named *AND fence*. This term is inspired by [29], where the term *Boolean fence* is used to refer to the topology of a logic network. In contrast, we use AND fence to describe the usage of ANDs in an abstract XAG, i.e., AND count, fanin size of each AND, and the step that each AND belongs to. This information is of particular interest to us because it directly and exclusively determines an abstract XAG's garbling cost. We denote each AND fence $\mathcal{F}$ as a set of integers:

$$\mathcal{F} = \{m_1, m_2, \cdots, m_p\}.$$

the i^{th} element indicates the fanin size of the i^{th}-step AND.

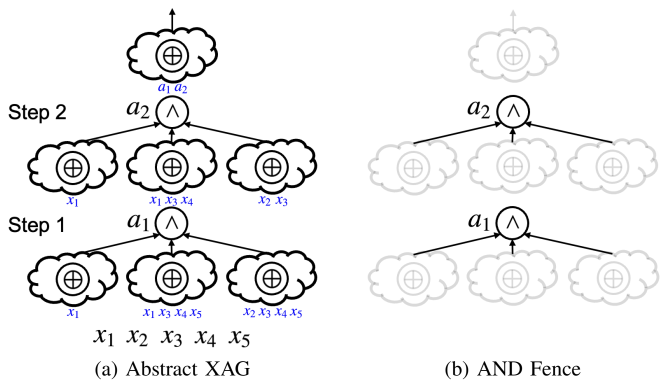


Fig. 4: An Abstract XAG Implementation of Function #$2888a000$ and its AND Fence

For example, Fig. 4a provides an abstract XAG implementation of the function we have seen before. For conciseness, instead of arrows, we use the blue texts under each XOR cloud to denote the fanins of that XOR cloud. By ignoring the XOR clouds, the AND fence of this abstract XAG is obtained (Fig. 4b), whose numerical representation is $\mathcal{F} = \{3, 3\}$, as both a_1 and a_2, the first-step and second-step ANDs, are 3-input.

Based on the AND fence of an abstract XAG, its garbling cost can be calculated follwing Eq. 1 given in Section III. An abstract XAG's structural MC can also be learned from its AND fence: since each m-input AND can be decomposed into $m - 1$ AND2s, an abstract XAG whose AND fence is $\mathcal{F} = \{m_1, m_2, \cdots, m_p\}$ is with the structural MC of

$$smc(\mathcal{F}) = \sum_{i=1}^{p}(m_i - 1) = \sum_{i=1}^{p} m_i - p \qquad (3)$$

B. Finding Optimal Abstract XAG

The fact that an abstract XAG's garbling cost can be learned from its AND fence has enabled us to efficiently find the optimal abstract XAG implementations for given functions.

Given a function, its optimal abstract XAG implementation can be found by enumerating potential AND fences in garbling cost ascending order. For each AND fence, we rely on the SAT solver to answer the following question: "Is there an abstract XAG that follows the given AND fence and implements our target function?" If the answer is yes, we manage to find an abstract XAG implementation for the function — since an AND fence with the lower garbling cost is always considered earlier, the optimality of the implementation is guaranteed; otherwise, the next AND fence would be targeted.

To support this proposal, a library of AND fences is needed. To take all fences of interest into consideration, our solution starts by addressing its simpler variant: How many AND fences are there when a certain structural MC is targeted? This is the reverse of calculating the structural MC of an AND fence (Eq. 3) and is essentially a *positive integer partition* problem, which is a classic problem in number theory and combinatorics. The permutation of numerically-different elements matters in our case, e.g., if $m_j \neq m_k$, $\{m_j, m_k\}$ and $\{m_k, m_j\}$ are two different AND fences.

We adapt the algorithm in [30] to our implementation. Since our goal is to generate a database for 6-variable Boolean functions and it is known that the functional MC of up-to-6 variable functions is upper-bounded by 6 [16], our library consists of AND fences whose structural MC is no more than 6. By executing the adapted algorithm, it is learned that there are 63 AND fences in total.

On top of such a library of AND fences, we propose a novel approach to efficiently find optimal abstract XAG implementations.

Algorithm 2: Efficiently Finding Optimal Abstract XAG using AND Fences

```
Input: Boolean function f, library of AND fences
Output: Optimal abstract XAG implementation for f
1 fmc ← functional MC of f
2 foreach AND fence $\mathcal{F}$ ∈ library do
3     s ← smc($\mathcal{F}$)
4     if $s < fmc$ then continue
5     N ← exact_synthesis(f, $\mathcal{F}$)
6     if $N \neq NULL$ then return N
```

The functional MCs of small-scale Boolean functions, whose numbers of input variables are no more than 6, are known [16]. Making use of a function's functional MC, the synthesis procedure is accelerated by skipping those AND fences providing an insufficient structural MC (line 4).

C. A Novel Approach to Synthesize Optimal X1G Efficiently

By integrating the proposed optimal abstract XAG synthesis technique (Algo. 2) and the one-to-one converting algorithm transforming a compact XAG into a lower-cost X1G (Algo. 1), we propose a novel approach to efficiently synthesize optimal X1G implementations.

Given a Boolean function, its garbling-cost optimal X1G implementation is found in three steps: (1) find the optimal abstract XAG implementation following Algo. 2; (2) transform the abstract XAG into a compact XAG by decomposing XOR clouds into XOR2s; (3) apply Algo. 1 to convert the compact XAG into an X1G.

As pointed out in Section III, for the same function, its optimal compact XAG and X1G implementations can be not interchangeable by applying Algo. 1 due to structural difference. Nevertheless, based on the AND fence of an abstract XAG, we can use Eq. 2 to "predict" what its garbling cost would be if it is converted into an X1G. Thus, when sorting the candidates in the library of potential AND fences, the sort criteria is the garbling costs calculated following Eq. 2. Put another way, for those "optimal abstract XAG implementations" synthesized in the first step, the optimality is defined in terms of the "predicted" garbling cost. For this reason, the first step can be also termed as a *conversion-aware optimal abstract XAG synthesis* process.

We recall the example shown before to demonstrate the effectiveness of this approach. When the target function is with the truth table of $^{\#}2888a000$, at least two OneHot3s are required to implement it as an X1G, as evidenced by Fig. 3b. To reach such an X1G implementation through applying Algo. 1, the starting point shall be a compact XAG whose AND fence is $\mathcal{F}_1 = \{3, 3\}$. If we focus on synthesizing optimal abstract XAG in the first step, the solver would be stuck at an implementation with the AND fence of $\mathcal{F}_2 = \{2, 2, 2\}$, as it achieves the minimum structural MC. Following the proposed approach however, it is foreseen that $\mathcal{F}_1$ would result in an X1G of lower garbling cost than $\mathcal{F}_2$ would do, as $cost_{X1G}(\mathcal{F}_1) = 4$ and $cost_{X1G}(\mathcal{F}_2) = 6$ according to Eq. 2; thus, it considers $\mathcal{F}_1$ prior to $\mathcal{F}_2$ and successfully finds the abstract XAG corresponding to the optimal X1G implementation. Specifically, the abstract XAG in Fig. 4a is exactly the one found by the proposed approach.

VI. Experimental Results

We here report the experimental evaluations on (a) the novel approach to synthesize optimal X1G implementations for small-scale functions; and (b) the database-driven Boolean rewriting-based X1G optimization method. All the experiments are run on an Apple M1 Max chip with 32GB memory.

A. X1G-Oriented Exact Synthesis Approach

Among different SAT encoding strategies for exact synthesis in the literature, *single selection variable* (SSV) is a widely welcomed one and selected as the object of comparison [21]. In order to provide a convincing evaluation, for the competitor we further adopt the heuristic of using the target function's functional MC to limit the ranges of the expected number of OneHot3s and XORs [12], which can considerably prune the search space for the optimal X1G implementation.

In this experiment, the benchmark consists of all representative functions of 48 affine equivalence classes for 5-variable Boolean functions. The exact synthesis solvers in both two approaches are implemented by exploiting the C++ reasoning library *bill*[2], with *Glucose*[3] adopted as the underneath SAT solver. The conflict limit for the SAT solver is set to 100 000.

[2]Available at: https://https://github.com/lsils/bill

[3]Available at: https://www.labri.fr/perso/lsimon/research/glucose/

TABLE I: Evaluation on 48 Representatives of 5-variable Affine Equivalence Classes

Synthesis method	#solutions	Accum. #OneHot3s	Accum. Time [s]
baseline	43	113	1 354.58
ours	48	120*	40.23

* This figure goes down to 105 if we exclude the 5 cases where no solutions are synthesized by the baseline method.

According to experimental results, the X1G implementations synthesized following the proposed method (ours) are always of equal or higher quality, i.e., require fewer OneHot3s, together with a 33.67× speed-up on average. Among the 48 target functions, the baseline method failed to find any solution for 5 of them, while the proposed method managed to synthesize X1G implementations for all target functions. For 7 functions, the solutions found by the baseline method are sub-optimal, as the proposed method found solutions using fewer OneHot3s, evidenced by the fact that the total number of OneHots (Accum. #OneHot3s) in the 43 solutions found by the baseline method is higher than the one in the corresponding 43 X1Gs synthesized following the proposed method.

With a reasonable number of conflict limits, the baseline method even failed to find the optimal X1G implementations for all these 48 functions. This observation implies the infeasibility of relying on conventional methods to achieve our goal of generating a database for 6-variable Boolean functions.

By capturing a significant characteristic of this practical GC generation problem that XORs are free of garbling cost, the proposed approach smartly removes the constraint on XOR count, so as to increase the number of solutions in the search space and make it possible to find a solution efficiently. By exploiting this approach, a 150 357-entry database of the garbling cost-optimal X1G implementations for 6-variable Boolean functions is generated, on which our database-driven Boolean rewriting-based X1G optimization method relies.

B. Database-Driven Rewriting-Based X1G Optimization

We implemented a Boolean rewriting-based X1G optimization method as part of the C++ logic synthesis library *mockturtle*[4]. For the usage of the database, we empirically set the maximum number of affine operations allowed for matching to be 100 000. For practical purposes, if a partial network's function cannot be matched to any of the representatives within this limitation, the rewriting targeting it would be skipped.

Three benchmark suites are targeted to provide a comprehensive evaluation: EPFL [31], cryptographic[5], and MPC benchmark suites [32]. The proposed method is run repetitively for each benchmark until no improvement in garbling cost reduction is observed.

Due to space limitations, we include only the detailed results on the EPFL benchmark suite of random/control circuits, as the state-of-the-art X1G optimization algorithm claimed to have achieved significant garbling cost reduction on them [12].

[4]Available at: https://https://github.com/lsils/mockturtle

[5]Available at: https://homes.esat.kuleuven.be/˜nsmart/MPC/

TABLE II: Evaluation using EPFL Benchmark Suite of Random/Control Circuits

Benchmark	Initial circuit	State of the art ($k = 5$) [12]			proposed optimization method ($k = 6$)					
	Garbling cost	Garbling cost	Red. [%]	Time [s]	#ite.	Garbling cost	Red. [%]	Time [s]	Impr. [%]	Speed-up
round-robin arbiter	2 348	1 658	29.39	2 857.40	3	1 522	35.18	17.54	8.20	162.91×
coding-cavlc	788	556	29.44	8 034.96	2	552	29.95	8.82	0.72	910.99×
ALU control unit	90	76	15.56	938.29	2	76	15.56	0.53	0.00	1 770.36×
decoder	656	656	0.00	0.01	1	656	0.00	0.01	0.00	1.00×
i2c controller	1 114	942	15.44	9 653.93	3	932	16.34	8.62	1.06	1 119.95×
int to float converter	170	140	17.65	2 306.66	2	138	18.82	1.96	1.43	1 176.87×
memory controller	9 390	7 578	19.30	34 950.05	5	7 456	20.60	48.74	1.61	717.07×
priority encoder	646	544	15.79	1 664.38	2	544	15.79	5.82	0.00	285.98×
look-ahead XY router	186	120	35.48	1 526.93	3	120	35.48	2.48	0.00	615.70×
voter	8 514	7 186	15.60	29 038.04	4	7 148	16.04	25.48	0.53	1 139.64×

Table II includes following information: garbling cost, which is the number of cipher-texts required to garble a OneHot3 gate times the number of OneHot3s in an X1G (i.e., 2·#OneHot3s), the reduction in garbling cost compared to the initial circuits (Red.), the number of iteration run until saturation (#ite.), the runtime (Time), the improvements in garbling cost reduction (Impr.) and speed-up (Speed-up) achieved by the proposed method over the state-of-the-art.

Without the need to invoke exact synthesis on the fly, a significant improvement in runtime is achieved by the proposed method. Moreover, for 6 out of the 10 benchmarks in Table II, the proposed method further managed to improve the reduction in garbling cost.

TABLE III: Summary of Comparison on All Benchmark Suites

Bench. suites	#improved bench.	Avg. impr. [%]	Avg. speed-up
EPFL	8 (20)*	1.97	2 441.64×
cryptographic	2 (10)	1.59	795.58×
MPC	6 (20)	0.51	3 950.08×

* Number in parentheses indicates the total number of involved benchmarks that belong to this benchmark suite.

Table III provides an overview of the performance of our X1G optimization method on all three adopted benchmark suites compared to the state-of-the-art, including the number of benchmarks on which an improvement in garbling cost reduction is achieved (#improved bench.), the average improvement in garbling cost reduction (Avg. impr.), and the average speed-up (Avg. speed-up).

The speed-up achieved by the proposed optimization method is consistent in all involved benchmarks. Together with a reduced runtime, an improvement in garbling cost reduction is realized in many benchmarks. This is credited to the generated database of the optimal X1G implementations for 6-variable Boolean functions: with a larger k ($k = 6$), the Boolean rewriting algorithm can focus on a larger portion of the logic network at a time and seize those optimization opportunities that are likely missed by the state-of-the-art, where a smaller k is adopted ($k = 5$) due to the poor scalability of exact synthesis.

For other benchmarks where no improvements in garbling cost reduction are achieved by our method, the resulting garbling cost is never higher than the state-of-the-art, except for two benchmarks. On *barrel shifter* in EPFL benchmark suite and *sha-256* in cryptographic benchmark suite, we have respectively witnessed a 0.02% and 0.04% higher garbling cost compared to the state-of-the-art. Through a case study, we learned that it is because of the constraint on the allowed maximum number of affine operations during the function matching procedure, due to which some chances for optimization would be given up. This problem can be addressed by loosening this constraint, with the risk of a potential increase in runtime.

VII. Conclusion and Discussion

Compared to XAG, both compact XAG and X1G are believed to be more appropriate choices of logic representations for the task of practical GC generation. In this paper, we proved that X1G is the better candidate, as evidenced by the fact that for any compact XAG, there always exists a functionally-equivalent X1G implementation with equal or lower garbling cost. Although the state-of-the-art X1G optimization algorithm achieved a significant reduction in garbling cost, it typically suffers from a prohibitive runtime, as it relies on invoking exact synthesis in an on-the-fly manner. To provide a more performant X1G optimization technique, we made two proposals: (a) generating a database of optimal X1G implementations by adopting the affine equivalence theory as the basis for Boolean function classification; and (b) a novelly-formulated exact synthesis approach to efficiently synthesize X1G implementation for small-scale functions. On top of the two proposals, we managed to generate a database of the optimal X1G implementations for 6-variable Boolean functions, which facilitates a database-driven cut rewriting-based X1G optimization method. A comprehensive experimental evaluation on three popular benchmark suites shows that our method on average outperforms the state-of-the-art algorithm in garbling cost reduction by respectively 1.97%, 1.59%, and 0.51%, together with 2441.64×, 795.58×, and 3950.08× speed-ups. The proposed method provides a powerful and agile technique for low-cost GC generation, enabling practical high-performance secure computation.

Acknowledgment

This project is supported in part by Synopsys Inc. We thank anonymous reviewers for their insightful comments.

References

[1] P. Bogetoft, I. Damgård, T. Jakobsen *et al.*, "A practical implementation of secure auctions based on multiparty integer computation," in *International Conference on Financial Cryptography and Data Security*. Springer, 2006, pp. 142–147.

[2] M. R. Clarkson, S. Chong, and A. C. Myers, "Civitas: Toward a secure voting system," in *IEEE Symposium on Security and Privacy*, 2008, pp. 354–368.

[3] F. Chen, X. Jiang, S. Wang *et al.*, "Perfectly secure and efficient two-party electronic-health-record linkage," *IEEE internet computing*, vol. 22, no. 2, pp. 32–41, 2018.

[4] 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.

[5] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[6] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[7] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008, pp. 486–498.

[8] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.

[9] M. Ball, T. Malkin, and M. Rosulek, "Garbling gadgets for boolean and arithmetic circuits," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 565–577.

[10] E. Testa, M. Soeken, H. Riener *et al.*, "A logic synthesis toolbox for reducing the multiplicative complexity in logic networks," in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2020, pp. 568–573.

[11] H.-L. Liu, Y.-T. Li, Y.-C. Chen *et al.*, "A don't-care-based approach to reducing the multiplicative complexity in logic networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4821–4825, 2022.

[12] M. Yu and G. De Micheli, "Generating lower-cost garbled circuits: logic synthesis can help," in *IEEE International Symposium on Hardware Oriented Security and Trust*, 2023.

[13] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 503–513.

[14] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 784–796.

[15] J. Boyar, P. Matthews, and R. Peralta, "Logic minimization techniques with applications to cryptology." *J. Cryptol.*, vol. 26, no. 2, pp. 280–312, 2013.

[16] Ç. Çalık, M. Sönmez Turan, and R. Peralta, "The multiplicative complexity of 6-variable boolean functions," *Cryptography and Communications*, vol. 11, no. 1, pp. 93–107, 2019.

[17] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *Proceedings of the 1st ACM Conference on Electronic Commerce*, 1999, pp. 129–139.

[18] C. Carlet, Y. Crama, and P. L. Hammer, "Boolean functions for cryptography and error-correcting codes." 2010.

[19] M. A. Harrison, "Introduction to switching and automata theory," *McGraw-Hill series in systems science*, 1965.

[20] M. Turan Sönmez and R. Peralta, "The multiplicative complexity of boolean functions on four and five variables," in *Lightweight Cryptography for Security and Privacy*. Springer, 2015, pp. 21–33.

[21] M. Soeken, W. Haaswijk, E. Testa *et al.*, "Practical exact synthesis," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 309–314.

[22] S.-Y. Lee and G. De Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[23] W. Haaswijk, M. Soeken, L. Amarú *et al.*, "A novel basis for logic rewriting," in *22nd Asia and South Pacific Design Automation Conference*. IEEE, 2017, pp. 151–156.

[24] E. Testa, M. Soeken, L. Amarù *et al.*, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[25] G. Meuli, M. Soeken, M. Roetteler *et al.*, "Enumerating optimal quantum circuits using spectral classification," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[26] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based fpgas," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, 2006, pp. 41–49.

[27] D. M. Miller and M. Soeken, "An algorithm for linear, affine and spectral classification of boolean functions," in *Advanced Boolean Techniques*. Springer, 2020, pp. 195–215.

[28] M. Soeken, "Determining the multiplicative complexity of boolean functions using SAT," *arXiv preprint arXiv:2005.01778*, 2020.

[29] W. Haaswijk, A. Mishchenko, M. Soeken *et al.*, "SAT based exact synthesis using DAG topology families," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[30] D. E. Knuth, *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees–History of Combinatorial Generation*. Addison-Wesley Professional, 2013.

[31] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis*, 2015.

[32] M. S. Riazi, M. Javaheripi, S. U. Hussain *et al.*, "Mpcircuits: Optimized circuit generation for secure multi-party computation," in *IEEE International Symposium on Hardware Oriented Security and Trust*, 2019, pp. 198–207.