Generating Lower-Cost Garbled Circuits: Logic Synthesis Can Help

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

***Abstract*—Garbled circuit (GC)-based frameworks are the cornerstone of advanced secure multi-party computation (MPC) protocols in various domains. These applications, such as secure network inference, require both scalability and real-time computation. However, the data communication among parties required by GC is currently a bottleneck of its runtime performance. Most existing works focus on minimizing the number of ANDs in logic networks over the basis {AND, XOR, NOT}, represented by XOR-AND graphs (XAG). AND is the only logic primitive among the three that contributes to providing the necessary multiplicative complexity (MC) of the desired logic function but causes communication costs. Inspired by the *garbling gadget* technique, we conduct a thorough study on the plausibility of adopting XAGs as the underneath logic representation to generate low-cost GCs and make two proposals: (1) merging small-fanin-size ANDs in XAGs, and (2) adopting OneHot gate, rather than AND, as the logic primitive to express MC, in order to reduce garbling costs. The first proposal optimizes GCs within a shorter runtime, whereas the second reduces garbling costs more. To validate our ideas, we propose a XAG-targeted merging algorithm and a logic synthesis flow for XOR-OneHot graphs (X1G). Compared to best-known results, our XAG- and X1G-targeted implementations achieve reductions in garbling cost by up to 25.27% and 35.48% respectively.**

***Index Terms*—garbled circuits, logic synthesis, multiplicative complexity**

I. Introduction

First proposed by Yao as an impactful protocol for secure two-party computation [1], *garbled circuit* (GC) allows two parties to use their private data as operands to jointly conduct arbitrary computation without revealing the content of each party's input to the other. With the advantages such as constant round complexity, GC is widely believed to be an ideal starting point to pursue performant *secure multi-party computation* (MPC). The famous *Beaver-Micali-Rogaway* (BMR) protocol is an example of a successful generalization of GC [2]. Meanwhile, the excessive communication among the involved parties required by GC-based protocols has seriously restricted the realization of low-latency and energy-efficient applications.

Nowadays, besides the well-known MPC tasks in the literature like secure auction [3] and voting [4], there is an increasing demand for privacy-preserving applications. For instance, in secure neural network inference, model providers avoid revealing their dedicatedly-trained network models, and the clients avoid leaking their sensitive input data [5]. Preferred solutions rely on a combination of cryptographical approaches. For example, *linearly homomorphic encryption* (LHE) is adopted for data encryption in linear layers, while GC guarantees the secure executions of non-linear activation functions [6]. It follows that improving the efficiency of GC is more urgent than ever before to meet these rapidly-increasing application requirements.

GC, as well as MPC protocols stemming from it, relies on representing the target function as a Boolean circuit. For the execution of each logic gate, cipher-texts have to be sent from the garbler to the executor by *oblivious transfer* (OT), which accounts for the mentioned communication complexity of GC-based protocols. Thus, reducing the communication cost (i.e., the number of cipher-texts to transfer) of each gate has a direct positive impact on the performance of applications. To achieve this, various efforts have been made. In particular, *free-XOR* [7] and *half-gate* [8] are commonly regarded as the major milestones. These two garbling schemes are compatible with each other, and a joint application enables the communication cost (hereafter referred to as garbling cost) of any 2-input logic gates to reach a proved minimum: 0 cost for XOR, XNOR and NOT gates; 2 cipher-texts for others. In other words, XOR, XNOR, and NOT gates are free in this context.

From the perspective of logic synthesis, the optimization objective of the aforementioned problem can be formulated as follows: Given a logic function implemented over the basis {AND, XOR, NOT} (which is functionally complete), reduce the number of AND gates as much as possible, since AND is the only logic primitive resulting in garbling cost. This problem is known as the minimization of the *multiplicative complexity* (MC) (i.e., multiplicative size) of Boolean circuits, or MC reduction. Great success has been achieved by the logic synthesis community: Both Songhori *et al.* and Riazi *et al.* base their works on establishing a customized library and exploiting standard logic synthesis tools to conduct library binding [9] [10]; Testa *et al.* create a logic synthesis toolbox, which leverages existing logic optimization techniques in a hybrid manner [11]; the most recent work by Liu *et al.* focuses on detecting the ANDs in a network that can be replaced by XNORs without changing the network's functionality [12].

On the other hand, experts in cryptography regard free-XOR and half-gate as near-optimal, if not the best, garbling schemes for Boolean circuits. Therefore, research focuses on figuring out a way to efficiently garble arithmetic circuits. For example, Ball *et al.* propose the idea of expressing complex arithmetic operations as mixed moduli circuits [13]. This so-called *gar-*

bling gadget technique suggests that symmetric gates can be garbled more efficiently, which is not yet exploited by the logic synthesis community. Inspired by garbling gadget, we take a fresh look at this goal of generating lower-cost garbled circuits, as it is currently limited by the intractability of the MC reduction problem.

In this paper, we propose to:

- merge the 2-input AND gates in XAGs into wider ANDs. Experimental results show that our devised merging algorithm reduces garbling costs by up to 25.27% within trivial runtime.
- use OneHot gates, instead of ANDs, as the logic primitive to express MC. A OneHot gate is more MC-expressive than an AND gate, while their garbling costs are the same. Extensive evaluations have proved the effectiveness of our *XOR-OneHot graph* (X1G)-targeted cut-rewriting and exact-synthesis logic optimization flow, as a reduction in garbling cost by up to 35.48% is achieved.

II. Preliminaries

A. Multiplicative Complexity

Given a logic function, its MC refers to the minimum number of AND gates required to implement this Boolean function over the basis $\{$AND, XOR, NOT$\}$ [14], or rather, as a *XOR-AND graph* (XAG), where complementation is indicated by edges and the other two kinds of gates are distinct nodes. We shall term the MC of a logic function as its *functional multiplicative complexity* (FMC), so as to distinguish it from the *structural multiplicative complexity* (SMC) of the function, which indicates the number of AND gates in an arbitrary XAG representing that function. Obviously, for any function, its FMC serves as the lower bound of the SMC of any XAG representation of this function.

Finding the FMC of a given logic function is a research topic receiving considerable attention, as it correlates to many fields: security [15], quantum computing [16] and more.

So far, the FMC of any logic function with no more than 6 inputs is known, but figuring out the FMC of a Boolean function is generally an intractable problem [15]. Thus, many works focus on logic functions that either exhibit certain characteristics, such as symmetric functions [17], or appear frequently in certain applications of interest, such as interval checking [18], etc. MC reduction of an arbitrary function can be addressed heuristically by: starting from a XAG, use logic manipulation to reduce the number of AND gates and get its SMC as close to the FMC of the function it represents as possible.

Fig. 1 is an example related to the 3-input majority function (whose output signal is true if only 2 or more input signals are true). Dotted lines indicate complemented edges. '$\wedge$' and '$\oplus$' respectively represent an AND gate and an XOR gate. While both XAGs correspond to the same logic function, the *3-input majority* (MAJ3), whose FMC is known to be 1, there is a difference between their SMCs: the SMC of the left-hand side is 3, while the SMC of the one on the right is 1. The latter

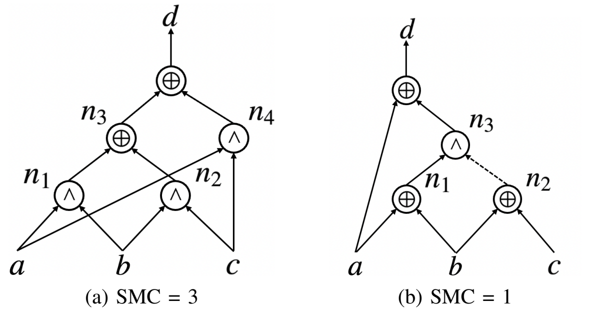


Fig. 1: Two XAGs representing MAJ3, with different SMCs

is the same as the FMC of MAJ3. Hence, in terms of MC reduction, it is reasonable to regard the right-hand side graph as an optimal XAG implementation of MAJ3.

B. Logic Synthesis

Logic synthesis generally refers to the process of converting a high-level described circuit into a lower-level representation, such as a gate-level netlist. It plays an important role in nowadays *electronic-design-automation* (EDA) flow, as it optimizes designs of integrated circuits for a specified cost criterion. The goal can typically be but is not limited to area, delay or power consumption, etc. The circuits processed by logic synthesis techniques are usually abstracted into directed graphs, often referred to as logic networks, for the ease of applying graph-based optimization algorithms to manipulate designs. The aforementioned MC reduction problem is essentially a logic synthesis problem.

Most logic synthesis techniques are heuristic, because the target logic networks are usually of large scale and complex, and the search spaces are too large to find the optimum solution in a reasonable time. A lot of logic synthesis techniques are based on the concept of *cuts* [19], which helps focus on a part of the target network at a time, such as the *cut rewriting* technique [20] that the X1G-targeted logic optimization flow presented in this paper heavily relies on. A cut is identified by its *root* and *leaves*, which is respectively a node and a collection of nodes. It contains all the nodes in the sub-graph bounded by its root and leaves. A feasible set of leaves shall meet two properties: there is at least one leaf on any path from a primary input to the root, and all the leaves are on at least one such path. A cut is said to be *k-feasible* if its number of leaves does not exceed k. For example, in Fig. 1b, the cut that consists of $\{b,\ c,\ n_1,\ n_2,\ n_3\}$ is 3-feasible, whose root is n_3 and leaves are $\{b,\ c,\ n_1\}$. Given a specified k, the process of finding all the k-feasible cuts in the target network is known as *cut enumeration* [19].

C. Garbled Circuits Generation

Garbled circuits generation refers to the process by which the garbler constructs a network of encrypted tables (i.e., a GC) based on a Boolean circuit/logic network representing the target computation. A logic network and the generated GC are isomorphic, as each logic primitive in the former corresponds to an encrypted table in the latter.

In a GC, secure computation is realized by: for any signal r in the original Boolean circuit, two labels X_r^0 and X_r^1 are to be created for the corresponding wire r in the GC, which indicate *false* and *true* in the original Boolean circuit respectively; each label is a bit-string, the length of which depends on the desired security level; each entry of an encrypted table is a cipher-text symmetrically encrypted using a potential combination of labels on the wires input to this table; since only one of the two labels on each wire is available to the executor, only one entry of each encrypted table can be correctly decrypted during the execution, preventing leakage of information. Generating a GC is essentially creating the aforementioned labels and encrypted tables.

While preserving privacy, the cipher-texts in encrypted tables are at the same time the source of communication cost, as they are to be sent from the garbler to the executor. Considering some kinds of logic gates are cipher-text-free, such as XORs [7], when synthesizing the original Boolean circuits, logic synthesis techniques can help generate lower-cost GCs by reducing the number of non-free gates. This is a stage prior to GC generation, on which our work focuses.

Optimally synthesizing Boolean circuits on the spot in an application-by-application manner can be time-consuming and may become another bottleneck of system performance. Instead, for those functions popular in secure computation, their Boolean circuits can be optimally synthesized in advance and made publicly available. There would not be any potential security liability, given that the security of the GC protocol comes from the encrypted tables.

D. Garbling Costs of Symmetric Functions

When garbling a Boolean circuit, adopting logic primitives with more fanins is hardly considered, because the garbling cost of a logic gate increases exponentially as the size of its fanin increases. As an example, the garbling cost of a *2-input AND* (AND2) is 4 (can be reduced to 2 by applying some advanced garbling schemes, such as half-gate), while a *3-input AND* (AND3)'s garbling cost is 8, due to the fact that there are 8 (2^3) potential input patterns.

However, garbling gadget [13] has yielded a fresh perspective on this problem. Intended to figure out a way to efficiently garble arithmetic circuits, garbling gadget has proposed the following observations: a *2-input XOR* (XOR2) operation is essentially a mod-2 addition, whose garbling cost is zero. Then, the definition of free-XOR can be generalized and applied to any m-input logic gate whose output depends only on the Hamming weight (the number of signals that are true among its m inputs) of its input pattern. This is because, such a logic operation, also known as a symmetric function, can be interpreted as a modular addition. The minimum modulus o is no more than $m+1$ (the calculation of o is described later) because the maximum Hamming weight of an input pattern of a m-input logic gate is m and the number of potential Hamming weights is therefore $m+1$ (ranging from 0 to m).

Therefore, for a symmetric logic gate, instead of garbling it in the conventional manner, which would result in a 2^m-entry encrypted table, a better solution to do the garbling is to interpret the operation as a mod-o addition followed by a projection gadget that encodes each sum back to the original finite field, GF(2). Since an o-to-2 projection gadget is garbled into a o-entry encrypted table, the garbling cost of such a m-input logic gate is now o, which is much better than in the baseline garbling scheme where it is 2^m.

Here is an example to compare the garbling of an AND3 gate (whose input signals are a, b, and c, and whose output signal is d) with the baseline garbling scheme adopted and with the garbling gadget technique applied:

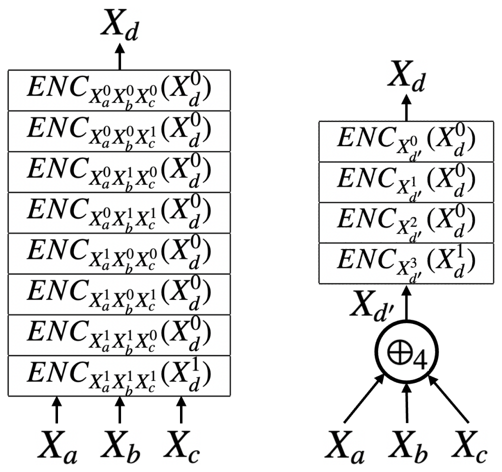

X_d
$ENC_{X_a^0X_b^0X_c^0}(X_d^0)$
$ENC_{X_a^0X_b^0X_c^1}(X_d^0)$
$ENC_{X_a^0X_b^1X_c^0}(X_d^0)$
$ENC_{X_a^0X_b^1X_c^1}(X_d^0)$
$ENC_{X_a^1X_b^0X_c^0}(X_d^0)$
$ENC_{X_a^1X_b^0X_c^1}(X_d^0)$
$ENC_{X_a^1X_b^1X_c^0}(X_d^0)$
$ENC_{X_a^1X_b^1X_c^1}(X_d^1)$
X_a X_b X_c
X_d
$ENC_{X_{d'}^0}(X_d^0)$
$ENC_{X_{d'}^1}(X_d^0)$
$ENC_{X_{d'}^2}(X_d^0)$
$ENC_{X_{d'}^3}(X_d^1)$
$X_{d'}$
$\oplus_4$
X_a X_b X_c
(a) The baseline scheme (b) Garbling gadget

Fig. 2: Garble an AND3 following different garbling schemes

For clarity, there is no entry permutation operated on the two encrypted tables in Fig. 2a and Fig. 2b.

As demonstrated before, based on the baseline garbling scheme, the fact that the truth table of a 3-input function is 8-entry determines that, after garbling, the corresponding encrypted table is 8-entry as well.

By contrast, when garbling gadget is applied, the operation of a symmetric logic gate is interpreted as a mod-o addition. Hence, it is possible that the potential number of labels on a wire exceeds two, and the output labels are no long strings of bits, but strings of integers. As shown in Fig. 2b, by regarding an AND3 function as a mod-4 addition (denoted as $\oplus_4$), for wire d', there are four potential labels (from $X_{d'}^0$ to $X_{d'}^3$); The projector would then operate a mapping, thanks to which the number of potential labels on each wire is still two and the result labels are still bit-strings. Such projectors guarantee that the execution of the remaining GC is valid.

Comparing the sizes of the two encrypted tables in Fig. 2a and Fig. 2b, it is obvious that by adopting garbling gadget, the garbling cost of an AND3 gate is reduced from 8 to 4 (which can be further reduced through applying other compatible optimizing schemes).

III. Exploit Symmetric Three-Input Gates

According to the baseline garbling schemes before garbling gadget, the garbling cost of a logic gate increases exponentially with the increase of its fanin size. Such a constraint makes it impossible to adopt logic gates with more fanins as logic primitives when synthesizing garbled circuits. However, garbling gadget provides us with a new perspective on the garbling of symmetric logic gates. The following question remains open:"*Is AND2 still our one and only choice of logic primitive to express MC?*" We propose to bring 3-input logic gates into the scope.

Since garbling gadget is a generalization of free-XOR, XOR gates and NOT gates are still free after the adoption of the garbling gadget technique. As the FMCs of an XOR gate and a NOT gate are both 0, the problem is further simplified as: "*Among 3-input logic gates, is there an alternative that is capable of expressing MC more cost-efficiently than AND2?*"

A. MC Compactness

As a quantified metric to evaluate choices of logic primitives, we define *MC compactness* as the FMC of a logic primitive divided by its garbling cost. For example, the MC compactness of an AND2 gate is $1/2 = 0.5$, which is a baseline for later comparison. A logic primitive of higher MC compactness is preferable here, because to provide a certain amount of MC (i.e., the FMC of the target logic function), using more MC-compact primitives may likely result in a lower garbling cost.

It is observed that none of the 3-input logic gates has an FMC larger than 2, as the FMCs of all 3-input gates are known [14]. Therefore, it is assured that no asymmetric logic gate has a preferable MC compactness due to the expensive garbling cost and should be excluded from our consideration. As a result, we are only interested in symmetric functions, to which garbling gadget is applicable.

B. A Study on Symmetric Three-Input Gates

Logic functions that can be mutually transformed into each other through applying input negation, input permutation and output negation are *negation-permutation-negation* (NPN)-equivalent and belong to the same NPN class [21]. All the 256 (2^8) 3-input Boolean functions are split into 14 NPN classes, 10 of which are non-trivial [22]. Regarding these 10 classes as 10 representative 3-input logic gates, the following four symmetric gates are of particular interest: AND3, MAJ3, OneHot (whose output is *true* if one and only one input signal is *true*), and Gamble (whose output is *true* only if the three input signals are all *true* or all *false*).

To calculate the MC compactness of these four candidates, their exact garbling costs are required. Using garbling gadget, each 3-input logic gate is interpreted as a mod-o addition followed by a o-to-2 project gadget, thus its garbling cost is o. To determine o for each candidate, i.e., the modulus to express each candidate as a modular addition, a simple algorithm is conceived based on the definition of modular additions.

Algorithm 1: Determine the modulus to express a given symmetric logic gate as a modular addition

```
Input: $m$-input function, $f$
Output: modulus to express the target function as a
        modular addition ($o$)
1 for $i \leftarrow 2$ to $m$ do
2     for $j \leftarrow i$ to $m$ do
3         if $f(\ HMW(j)\ ) \neq f(\ HMW(j \bmod i)\ )$ then
4             break
5         if $j = m$ then
6             return $i$
7 return $m$+1
```

In Algo. 1, *HMW(i)* denotes those input patterns whose Hamming weights are i. For example, when m is specified as 3, then *HMW(1)* refers to the following three input patterns: 001, 010, and 100.

With Algo. 1, the specific garbling cost of any m-input symmetric logic gate under garbling gadget can be calculated, as shown in Table I. Notice that, since garbling gadget is compatible with another technique, *garbled row reduction* [23], the application of which can further reduce the eventual garbling cost from o to $o - 1$.

TABLE I: The characteristics of the four three-input gates of interest and AND2 gate

Gate type	Truth table	FMC	Garbling cost	MC compactness
AND2	0x8	1	2	0.5
AND3	0x80	2	3	0.67
MAJ3	0xe8	1	3	0.23
OneHot	0x16	2	2	1
Gamble	0x81	1	2	0.5

In the column labeled "*Truth table*", the truth table of each logic gate is represented in hexadecimal as a bit-string, with the most significant bit on the left-hand side.

While both AND3 and OneHot are more MC-compact than AND2, OneHot gate possesses the highest MC compactness. But how to take advantage of this insight is not immediately apparent. This is because, the available literature mainly focuses on developing logic optimization techniques to reduce the number of AND gates after representing the target function as a XAG, where the allowed logic gates are AND2, XOR2, and NOT gates (*AND2 reduction* for short) [11] [12]. Thus, we at first explore a way to bridge our findings and the existing works on AND2 reduction.

IV. Merge Small-Fanin-Size ANDs

According to the proposed Algo. 1, to figure out the garbling cost of a symmetric gate with an arbitrary fanin size is trivial. The garbling cost of an m-input AND gate is m (with garbled row reduction applied as well). Meanwhile, based on the definition of MC, its FMC is $m - 1$. Its MC compactness is, therefore, $(m - 1)/m$. In other words, compared to

AND2, wider ANDs are more MC-compact and are therefore preferable as the logic primitives for low-cost garbled circuit generation. Hence, we propose to relax the restriction on the fanin size of AND gates.

We are hereafter manipulating graphs, where the AND gates and XOR gates are nodes, and NOT gates are represented as an attribute of edges. In this context, the terms AND/XOR gates and AND/XOR nodes are used interchangeably. We regard this task of replacing groups of adjacent AND2 nodes with nodes indicating wider AND gates as a post-processing step on optimized XAGs (where all the AND nodes denote AND2 gates). The benefit is twofold: we can make full use of the existing knowledge on AND2 reduction, and the runtime of such a post-process is smaller than synthesizing or restructuring a network. This approach is an alternative to resynthesizing from scratch the logic network in terms of wider ANDs and is more effective to realize.

An algorithm to merge ANDs nodes is to be elaborated, in order to reduce the garbling costs of XAGs as much as possible while keeping their functions unchanged.

A. To Merge or not to Merge

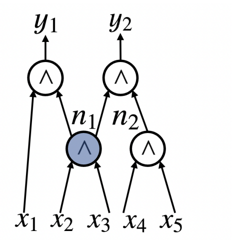
The question to answer is how to detect the largest possible groups of AND2 nodes that can be merged into wider AND gates. In each such group of AND2 nodes, we call the node that is closest to the primary output side the root node of this group, and term the opposites as leaf nodes — similar to the definitions of root and leaves when introducing cut.

To determine if an AND2 node is a root node, some criteria are evident: an AND2 is a root node if

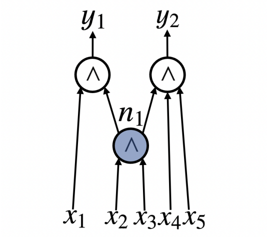
1) it is a primary output of the logic network;
2) it connects to an XOR node, or is followed by a negated edge, i.e., directly contributes to a non-AND2 gate (XOR or NOT).

Meanwhile, there is uncertainty when an AND2 is followed by more than one AND2. Such an AND2 indicates that a local function is shared by several groups and therefore shall not be merged into any one of them, as this shared logic would otherwise become unavailable to the remaining groups.

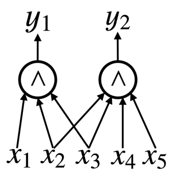
Node n_1 of the logic network in Fig. 3a, marked in blue, is such an example, as it is contributing to both node y_1 and Node y_2, which are AND2s. Fig. 3b shows that, if we leave node n_1 untouched during the merging procedure, the result garbling cost of the network is $2+2+3 = 7$. It is also possible to operate a more aggressive merging, with the precondition of duplicating node n_1 once, and obtain a network that looks further compact, whose garbling cost is $3 + 4 = 7$ as well, as shown in Fig. 3c. Both these two solutions manage to achieve lower-cost garbled circuits, since the original garbling cost is $2 \times 4 = 8$. But this does not suggest that the two options are always equally good — hereafter, we formally prove that, while merging is generally preferable, the style of merging matters. Operating aggressive merging can lead to a sub-optimal result, especially nodes like n_1 in this example shall not be merged.



(a) The original logic network



(b) Leave n_1 not merged



(c) Merge n_1 after duplication

Fig. 3: A logic network under different merging strategies

Assume that several groups of AND2 nodes, a_1, a_2, $\cdots$, a_p ($p \geq 2$), share and share only one sub-group of AND2s, a_{share}. Then, different merging strategies allow us to either duplicate a_{share} for q times and merge them into any q ($q+1$, when $q = p-1$) of these p groups, or leave a_{share} not merged. We abbreviate the two merging strategies to *merge shared* ($1 \leq q \leq p - 1$) and *keep shared* ($q = 0$).

Proposition 1. *For the same logic network, the garbling cost corresponding to the "merge shared" merging strategy is always better than or equal to the "keep shared" one.*

Proof. Denote the number of AND2 nodes in the group a_i as and_i, then gc_i, the garbling cost of a_i, is numerically equal to $and_i + 1$.

Without loss of generality, it is assumed that: among the p groups, the first q ($q + 1$ when $q = p - 1$) groups, a_1, a_2, $\cdots$, a_q, are merged with the sub-group a_{share} kept in them; By contrast, the remaining $p - q$ (0 when $q = p - 1$) groups have to be merged with the sub-group a_{share} spun off, i.e., $a'_i = a_i \backslash a_{share}$, whose garbling costs are denoted as gc'_i.

Then, we can use p and q to represent the garbling cost of the logic network after merging. When $q < p - 1$:

$$\begin{aligned} gc &= \sum_{i=1}^{q} gc_i + \sum_{i=q+1}^{p} gc'_i + gc_{share} \\ &= \sum_{i=1}^{q} (and_i + 1) + \sum_{i=q+1}^{p} (and_i - and_{share} + 1) + (and_{share} + 1) \\ &= \sum_{i=1}^{p} and_i - (p-1) \cdot and_{share} + q \cdot and_{share} + p + 1 \end{aligned} \tag{1}$$

The choice in merging strategy is essentially the choice in the value of q. Since and_{share} is at least 1 — if such a shared sub-group exists, it must consist of at least one AND2 node

— Eq. 1 would be minimized if q is minimized (i.e., assigned to 0), which refers to the *keep share* strategy:

$$\begin{aligned} gc_{keep\ share} &= gc|_{q=0} \\ &= \sum_{i=1}^{p} and_i - (p-1) \cdot and_{share} + p + 1 \end{aligned} \tag{2}$$

When *merge share* is to be adopted, q equals $p-1$. Since all the p groups would be merged with the shared sub-group a_{share} kept in them, the garbling cost of the network reduces to the sum of the garbling costs of these p groups. The gc under the current situation becomes:

$$\begin{aligned} gc|_{q=p-1} &= gc_{merge\ share} \\ &= \sum_{i=1}^{p} gc_i = \sum_{i=1}^{p}(and_i + 1) = \sum_{i=1}^{p} and_i + p \end{aligned} \tag{3}$$

Considering that $p \geq 2$ and $and_{share} \geq 1$, the difference between $gc_{merge\ share}$ (Eq. 3) and $gc_{keep\ share}$ (Eq. 2) never goes negative: since it is a positive correlation with both p and and_{share}, it reaches the minimum (0) when p and and_{share} are respectively assigned to 2 and 1 (the example in Fig. 3 illustrates exactly such a case). QED

Proposition 1 indicates that an AND2 shall be regarded as a root node if it is directly followed by more than one AND2 node. This criterion, together with the two proposed at the beginning of this sub-section, constitutes our complete standard to judge whether an AND2 node is the root node of another group or belongs to an existing group.

With root nodes successfully determined, the border of each group of AND2s becomes obvious, eliminating the necessity of figuring out leaves nodes.

B. An Exact Algorithm for Merging

We present an exact algorithm for merging: given a topologically ordered XAG, where all the AND nodes are 2-input, the algorithm deterministically detects all the opportunities to merge groups of AND2s into nodes representing wider ANDs, so as to produce a more compact network that is of lower garbling cost. The previously proposed criteria for finding the root nodes of each AND2 group serve as the core of this algorithm. For clarity, in the psuedocode to be shown below, we represent the judgment on whether an AND2 node n is a root node according to the criteria, as *is_root(n)*.

The *merge* function in Algo. 2 takes two parameters: the node that is judged to be the root node, which later becomes the primary output of the produced larger-fanin-size AND, and a set of nodes that are recognized as the leaves of a group, serving as the inputs to this AND node.

The proposed Algo. 2 has a linear time complexity, as each AND2 node in the target XAG is to be judged only once whether it should be regarded as a root node.

Algorithm 2: To merge AND2 gates in XAGs into larger-fanin-size AND gates

```
Input: XAG, $xag$
Output: compacted XAG
foreach nodes $n \in xag$ in reverse topological order do
    if is_AND(n) and !is_marked(n) then
        $leaves \leftarrow$ null
        determine_group_recursive($n$,$n$)
        merge($n$,$leaves$)
return $xag$

Function determine_group_recursive(node,root) :
    if !is_marked(node) then
        if $node = root$ then
            mark($node$)
            foreach node $l \in$ fanins of $node$ do
                if !is_root(l) then
                    determine_group_recursive($l$,$root$)
        else
            if !is_root(l) then
                mark($node$)
                add $node$ into $leaves$
                foreach node $l \in$ fanins of $node$ do
                    if !is_root(l) then
                        determine_group_recursive($l$,$root$)
```

V. The Key to Lower-Cost GCs: One-Hot Gates

As previously pointed out, OneHot is recognized as the most MC-compact 3-input gate. In this section, we focus on how to make use of this observation to generate lower-cost garbled circuits.

A. The Most MC-Compact Logic Primitive

Because

$$OneHot(0,0,0) = OneHot(1,1,1) = 0,$$

and that

$$HMW(0) = \{(0,0,0)\}, HMW(3) = \{(1,1,1)\},$$

we have

$$OneHot(HMW(0)) = OneHot(HMW(3)).$$

Recall that *HMW(i)* refers to the input patterns whose Hamming weights are i. This feature enables a OneHot operation to be interpreted as a mod-3 addition. It is further noticed that the FMC of a OneHot operation is 2: representing the function in the *algebraic normal form* (ANF) [15], $abc \oplus a \oplus b \oplus c$, the expression indicates that at least 2 AND2 gates are required to implement it over the {AND, XOR, NOT} basis. Therefore, while Gamble has the first feature as well, these two features combine to result in OneHot's being the most MC-compact logic primitive.

Proposition 2. *Given a XAG, there is always an X1G that represents the same logic function but is of lower or equal garbling cost.*

Proof. According to the definition of MC, when implementing an arbitrary Boolean function f as a XAG, xag_f, whose SMC is smc_f, then there are smc_f AND2s in xag_f.

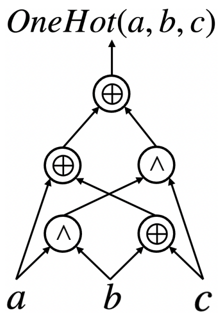


Fig. 4: A SMC-optimial XAG of the OneHot function

When the same function is represented as an X1G, we denote the number of OneHot gates required as #OneHot. Then, there must be at least one X1G whose #OneHot satisfies the inequality below:

$$\lceil \frac{fmc_f}{2} \rceil \leq \lceil \frac{smc_f}{2} \rceil \leq \#OneHot \leq smc_f \qquad (4)$$

Here is how we derived the upper and lower bounds in Inequality 4: There is a one-to-one correspondence between an AND2 gate and a OneHot gate, as

$$AND(a, b) = OneHot(1, \neg a, \neg b)$$

where "¬" indicates negation. Therefore, we can convert xag_f into a *XOR-OneHot graph* (X1G) by simply replacing each AND2 gate with a OneHot gate, which is the case indicating the upper bound of #OneHot. A convincing example is the situation where the target function f is exactly a OneHot operation: we can represent it as the XAG in Fig. 4, or a special X1G that consists of only 1 OneHot node. On the other hand, the fact that each OneHot gate inherently contains two AND2 gates, as shown in Fig. 4, implies the possibility to implement the same function in X1G using only $\lceil \frac{smc_f}{2} \rceil$ OneHot gates. Notice that any smc_f is lower-bounded by the FMC of the target function f, fmc_f, which accounts for the lower bound of the inequality. It is also obvious that this minimum is not always achievable, since the relative position of the 2 AND2 gates contained in each OneHot gate is fixed, which has undermined their expressiveness.

Considering that the garbling cost of a OneHot gate proves to be the same as an AND2 gate in previous sections, the right-hand part of Inequality 4 exactly proves the proposition. QED

Proposition 2 suggests using OneHot, instead of AND2, as the logic primitive to model the circuit when searching for lower-cost garbled circuits. Indeed, even in the worst situation where #OneHot reaches the upper bound (smc_f), the garbling cost in total would never exceed the baseline.

B. Cut-Rewriting & Exact-Synthesis Logic Optimization Flow

To support the proposal of adopting X1G over XAG as the underneath logic representation, we have elaborated a logic-optimization flow, which is heavily based on two logic synthesis techniques, *exact synthesis* and *cut rewriting*.

Algorithm 3: A cut-rewriting & exact-synthesis logic optimization flow targeting X1Gs

```
Input: XAG $xag$; cut size $k$
Output: X1G that is functionally equal to $xag$
1  $x1g \leftarrow$ one-to-one_transform($xag$)
2  $\#OneHot \leftarrow$ count_OneHot($x1g$)
3  do
4      $cuts \leftarrow$ compute_cuts($x1g$, $k$)
5      foreach cut $c \in cuts$ do
6          $f \leftarrow$ Boolean function of cut $c$
7          $fmc_f \leftarrow$ look_up_mc($f$)
8          for $\#OneHot' \leftarrow lower\_bound\_OneHot$ to $upper\_bound\_OneHot$ do
9              for $\#XOR3' \leftarrow lower\_bound\_XOR3$ to $upper\_bound\_XOR3$ do
10                 $c\_opt \leftarrow$ exact_syn($\#OneHot'$, $\#XOR3'$)
11                 if $c\_opt \neq$ null then
12                     rewrite cut $c$ in $x1g$ with $c\_opt$
13 while count_OneHot($x1g$) < $\#OneHot$
14 return $x1g$
```

Given allowed logic primitives, a target function, and specified cost criteria, exact synthesis finds the optimum implementation of the function [24]. However, the excessive computational complexity beneath makes this technique prohibitive and only applicable to small-scale functions. It is therefore preferable to make use of cut rewriting and exact synthesis in combination: the former allows us to focus on a small portion of the whole network at a time, and the latter is in the charge of providing the optimum solution required by the former to rewrite the sub-network [20].

To conduct exact synthesis, two parameters are required by the solver: the aimed numbers of OneHot gates, #OneHot', and *3-input XOR* (XOR3) gates, #XOR3', contained in the network to be synthesized. We adopt XOR3 over XOR2 so that the fanin size of the allowed logic primitives in the synthesis problem is consistent. Beware of the difference between the previously defined #OneHot and #Onehot': the former denotes the number of OneHot gates in a logic network, while the latter is merely a parameter for the exact synthesis solver, which might be unfeasible if the solver fails to find a solution under the given parameters; So is the difference between #XOR3 and #XOR3'.

It is necessary to figure out the FMC of the target function f, fmc_f, as its numerical relation between #OneHot is revealed in Inequality 4. We use the spectral classification-based approach — first, compute the Rademacher-Walsh spectrum of the target function, then use the obtained coefficients for classification [25] — to quickly lookup a function's FMC, which accounts for the *look_up_mc()* function in the pesudocode depicting our flow. With fmc_f obtained, #OneHot' is bounded by the range that Inequality 4 indicates. Additionally, #XOR3' is empirically set to be no more than twice the target number of non-free gates when the FMC is not zero, otherwise set to 2, as two XOR3s are sufficient to represent any function with zero FMC.

To leverage powerful XAG-targeted MC-reduction techniques in the literature, our flow takes an optimized XAG as input. Since we have pointed out that an AND2 gate can be replaced by a OneHot gate and $XOR2(a,b)$ equals $XOR3(0,a,b)$, we can trivially transform a XAG into an X1G by replacing each AND2/XOR2 node with a OneHot/XOR3 node. In the pesudocode, we denote this transformation as the *one-to-one_transform()* function.

VI. Experimental Results

Extensive experimental evaluations are conducted to validate our proposed ideas for lower-cost garbled circuit generation: (1) merging small-fanin-size ANDs in XAG, and (2) adopting X1G rather than XAG as the logic representation. The evaluations can be divided into two categories: a lightweight proof-of-concept that benchmarks small-scale functions, and results on practical benchmark suites.

Our implementations are on top of the logic network library *mockturtle*, and the exact synthesis solver is implemented exploiting the exact synthesis library *percy*, where *MiniSAT* [26] is the underneath SAT solver. Both of them belong to the EPFL logic synthesis libraries [27]. All the experiments are carried out on a machine having 256GB RAM and dual Intel Xeon E5-2680 v3 operating at 2.50 GHz.

A. Four-Input NPN Functions

Evaluation regarding 4-input Boolean functions as the target functions offers a peek into the effectiveness of these techniques. All 65536 (2^{16}) 4-input Boolean functions can be partitioned into 222 NPN classes. Since the FMCs of these functions are known [15], to each of them, exact synthesis is directly applied to generate the optimum implementations in both XAG and X1G. Optimum means to respectively use the minimum number of AND2 gates and OneHot gates in these two cases. More specifically, in this experiment, only line 6-12 in Algo. 3 are necessary, as we are synthesizing the whole X1Gs from scratch, instead of rewriting cut by cut.

TABLE II: Experimental results for 4-input NPN functions

Logic representation	#functions whose garbling costs are						Avg.
	0	2	3	4	5	6	garbling cost
XAG (baseline)	5	19	0	108	0	90	4.55
XAG (merged)	5	19	8	101	15	74	4.44
X1G	5	49	0	168	0	0	3.47

In Table II, there are no column counting functions whose garbling costs are one, as such a case is not possible: if a function's FMC is not zero, its garbling cost is at least two, which is the garbling cost of an AND2 gate or a OneHot gate. For the same reason, when XAG (baseline) or X1G is adopted as the logic representation, the result garbling cost is always even.

The effectiveness of the two proposed ideas disclosed by the experimental results is in line with our expectations. On the one hand, using X1G, rather than the baseline XAG, has effectively reduced the garbling costs by 23.74% on average; applying the XAG-targeted merging algorithm has also led to an average garbling cost reduction of 2.42% over the baseline. Notice that the FMC of any 4-input logic function is at most 3, which means the opportunities for merging AND2s are very rare in this experiment. Thus, the small gain obtained by merging is reasonable and does not invalidate our proposal, as can be learned later from the results on larger-scale benchmark suites. On the other hand, the runtime costs of the two proposals are significantly different, which is also consistent with our understanding: among the involved 222 cases, the merging procedure on average costs 3.14μs, and synthesizing optimum X1G takes 3.14s on average, approximately a million times longer than the former.

B. EPFL Benchmark Suite

Hereafter, we are comparing the gain in garbling-cost reduction achieved by the proposed algorithms to the state-of-the-art, which is collected from three works in the literature [10] [11] [12], as none of them is dominating the others in all the benchmarks. Since all these works are adopting XAG as the logic representation and have reported the numbers of AND2 gates in the XAGs optimized by their methodologies, #AND2, we reasonably regard corresponding garbling costs as 2·#AND2.

As for implementation details, we set the cut size, k, which is required by Algo. 3, to be 5, as we experimentally figure out that it strikes a good balance between garbling-cost reduction and runtime. Furthermore, we adopt the idea of *Boolean function mining*, first proposed in [28], as a trick to speed up the execution of Algo. 3: after the exact synthesis solver finds the optimum implementation for a cut, the solution would be stored in a cache and indexed by the function of this cut. In this way, if the function of a cut is recognized to be NPN equivalent to an entry of the cache, we can reuse this previously generated solution, instead of always relying on the time-consuming exact synthesis technique. We pass the XAGs optimized by existing works to the proposed two algorithms as input, except for the cases where the optimized benchmarks are not available. The aforementioned settings are applicable to our experiments on all three benchmark suites that are introduced below.

The EPFL benchmark suite consists of two kinds of combinational circuits, arithmetic ones and random/control ones [29]. Due to their distinguishing features, we have calculated the geometric means separately.

In the column labeled "*max. fanin*", we report the size of the largest group of AND2s that can be merged into a wider AND gate. Additionally, since our logic optimization flow for X1G keeps running until no more gains in garbling cost are obtained, as can be learned from the do-while loop (line 3-13) in Algo. 3, we present the number of iterations run in total in the "*ite.*" column.

TABLE III: Experimental results for EPFL benchmark suite

Benchmark	XAG (baseline)		XAG (merged)				X1G			
	garbling cost	source	max. fanin	garbling cost	impr.	time[s]	ite.	garbling cost	impr.	time [s]
adder	256	[11]	2	256	0.00%	<0.01	1	256	0.00%	2408.28
barrel shifter	1664	[11]	2	1664	0.00%	0.05	1	1664	0.00%	0.39
divisor	10264	[12]	5	10056	2.03%	1.74	3	9728	5.22%	11464.30
log2	17546	[12]	4	17249	1.69%	5.89	3	16358	6.77%	25234.87
max	1744	[12]	5	1692	2.98%	0.06	2	1656	5.05%	1901.11
multiplier	15170	[12]	4	15092	0.51%	2.95	4	14902	1.77%	6887.82
sine	3918	[12]	4	3810	2.76%	0.25	4	3480	11.18%	12393.11
square-root	10434	[12]	4	10296	1.32%	1.94	4	9942	4.72%	10662.73
square	9192	[12]	4	9121	0.77%	1.40	3	8890	3.29%	7010.90
geometric mean	**4503.95**			**4403.34**	**2.23%**			**4311.23**	**4.28%**	
round-robin arbiter	2348	[11]	5	2301	2.00%	0.02	3	1658	29.39%	2857.40
ALU control unit	90	[11]	5	85	5.56%	<0.01	2	76	29.44%	938.29
coding-cavlc	788	[11]	5	705	10.53%	<0.01	2	556	15.56%	8034.96
decoder	656	[11]	2	656	0.00%	<0.01	1	656	0.00%	0.01
i2c controller	1114	[11]	4	1048	5.92%	<0.01	3	942	15.44%	9653.93
int to float converter	170	[11]	3	157	7.65%	<0.01	2	140	17.65%	2306.66
memory controller	9390	[11]	16	8648	7.90%	0.63	2	7578	19.30%	34950.05
priority encoder	646	[11]	8	614	4.95%	<0.01	2	544	15.79%	1664.38
look-ahead XY router	186	[11]	9	139	25.27%	<0.01	3	120	35.48%	1526.93
voter	8514	[11]	5	8157	4.19%	1.15	5	7186	15.60%	29038.04
geometric mean	**850.80**			**785.65**	**7.66%**			**681.30**	**19.92%**	

As expected, the two proposed two algorithms always produce networks whose garbling costs are lower than, or equal to the starting point. Our second observation from Table III is that the performance of our two proposals are relevant: for those benchmarks that have plenty of opportunities to conduct merging, the reduction in garbling costs achieved by the X1G-targeted logic optimization flow is also generally considerable. For instance, both our XAG- and X1G-targeted methods performed best for the *look-ahead XY router* benchmark. We explain this observation in terms of the property of OneHot gates: according to Fig. 4, we know that the two AND2 gates contained in each OneHot gate are adjacent and can actually be merged into an AND3 gate if follow the proposed merging algorithm (Algo. 2). Hence, those cases where we can hardly make use of OneHot to achieve a lower garbling cost (e.g., *adder*, *barrel shifter* and *decoder*) are also the cases where chances of merging are very few, and vice versa.

We emphasize that our reported runtime shall not be confused with the generation time of garbled circuits, as illustrated in Section II-C.

C. Cryptographic and MPC Benchmark Suites

The two benchmark suites adopted in this sub-section respectively consist of cryptographic functions, such as block ciphers[1], and well-known MPC tasks, e.g., secure auction and stable matching problem [10].

According to Table V, there are higher-garbling-cost logic networks generated by our methods, such as *auction_N3_W16* and *stable matching_ks8_s8*, which is against our argument that our proposals are always contributing to lower-cost, at least equal-cost, compared to the starting points. This is because, while the best-known results for these cases are all reported by [10], the optimized benchmarks are not publicly available, and we, therefore, use the XAGs optimized by [11] instead as the starting points. For the same reason, there are cases where *ite.* is larger than 1 but with a zero or even negative *impr.*, such as the *auction_N3_W16* benchmark. In these cases, there is initially a significant gap between our starting points and our competing data. For instance, for the *stable matching_ks8_s8* benchmark, the garbling cost of our starting point, collected from [11], is 117446, 25.85% higher than the reported state-of-the-art. Applying our proposed XAG- and X1G-targeted algorithms have respectively achieved 6.48% and 9.84% improvements, considerably narrowing the gap. Thus, when referring to Table V, we suggest to use column "*max. fanin*" and "*ite.*", instead of "*impr.*", as the reference for judging whether the proposed algorithms have reduced the garbling cost of the starting point.

An analysis of the MPC circuits shows that AND gates are in general far from each other, and this is a likely explanation for the smaller effectiveness of our methods for these benchmarks, compared to other benchmark suites. Therefore, it is of interest to explore the possibility of integrating the preference for adjacent ANDs into the synthesis procedure.

VII. Conclusion

Existing efforts on reducing garbling costs focus on reducing the number of AND2 gates over XAGs, known as the MC reduction problem (more specifically, AND2 reduction problem). However, in this specific context of low-cost GC generation, it is an open question whether ANDs are preferable logic primitives to provide MC for target functions. Inspired

[1]Available at: https://homes.esat.kuleuven.be/~nsmart/MPC/

TABLE IV: Experimental results for cryptographic benchmark suite

Benchmark	XAG (baseline)		XAG (merged)				X1G			
	garbling cost	source	max. fanin	garbling cost	impr.	time[s]	ite.	garbling cost	impr.	time[s]
AES (Key Expansion)	10880	[11]	3	10560	2.94%	7.13	2	10240	5.88%	6825.42
AES (No Key Expansion	13600	[11]	3	13200	2.94%	10.16	2	12800	5.88%	15417.69
DES (Key Expansion)	13830	[12]	7	13479	2.54%	5.32	2	13144	4.87%	40162.40
DES (No Key Expansion)	13666	[12]	7	13340	2.39%	4.95	4	12984	4.99%	25369.60
MD5	18734	[11]	2	18734	0.00%	14.58	1	18734	0.00%	10936.36
SHA-1	22966	[12]	5	22873	0.40%	31.11	2	22688	1.21%	8225.44
SHA-256	52928	[12]	6	52128	1.51%	137.18	6	50540	4.51%	9942.18
Comp. 32-bit Signed LTEQ	174	[12]	5	159	8.62%	<0.01	3	140	19.54%	1135.75
Comp. 32-bit Signed LT	168	[12]	7	149	11.31%	<0.01	3	130	22.62%	1286.35
Comp. 32-bit Unsigned LTEQ	174	[12]	5	159	8.62%	<0.01	3	140	19.54%	1151.97
Comp. 32-bit Unsigned LT	168	[12]	7	149	11.31%	<0.01	3	130	22.62%	1254.59
geometric mean	**3322.25**			**3160.45**	**4.87%**			**2970.95**	**10.57%**	

TABLE V: Experimental results for MPC benchmark suite

Benchmark	XAG (baseline)		XAG (merged)				X1G			
	garbling cost	source	max. fanin	garbling cost	impr.	time[s]	ite.	garbling cost	impr.	time[s]
auction_N2_W16	194	[11]	2	194	0.00%	<0.01	1	194	0.00%	1327.60
auction_N2_W32	386	[11]	2	386	0.00%	<0.01	1	386	0.00%	1305.30
auction_N3_W16	456	[10]	2	464	-1.75%	<0.01	2	460	-0.88%	1468.07
auction_N3_W32	908	[10]	2	912	-0.44%	0.02	2	908	0.00%	1466.40
auction_N4_W16	984	[10]	2	990	-0.61%	0.02	2	986	-0.20%	1866.93
auction_N4_W32	1950	[11]	2	1950	0.00%	0.10	2	1946	0.21%	1873.25
NNS_K1_N8	1108	[11]	2	1108	0.00%	0.05	2	1104	0.36%	2490.47
NNS_K1_N16	2320	[10]	2	2324	-0.17%	0.21	2	2312	0.34%	2490.13
NNS_K2_N8	1762	[11]	4	1739	1.31%	0.13	2	1698	3.63%	2269.80
NNS_K2_N16	3838	[11]	4	3783	1.43%	0.57	2	3692	3.80%	2754.43
NNS_K3_N8	2120	[11]	3	2114	0.28%	0.21	2	2092	1.32%	2622.72
NNS_K3_N16	4788	[11]	3	4774	0.29%	1.04	2	4724	1.34%	3049.39
voting_N1_M3	14	[11]	2	14	0.00%	<0.01	1	14	0.00%	122.61
voting_N1_M4	30	[11]	3	29	3.33%	<0.01	2	28	6.67%	143.92
voting_N2_M2	42	[11]	3	41	2.38%	<0.01	2	40	4.76%	1007.06
voting_N2_M3	110	[11]	2	110	0.00%	<0.01	1	110	0.00%	1590.82
voting_N2_M4	208	[11]	2	208	0.00%	<0.01	1	208	0.00%	2077.66
voting_N3_M4	550	[11]	2	550	0.00%	<0.01	3	546	0.73%	2231.78
stable matching_ks4_s8	32002	[11]	13	30040	6.13%	7.87	3	28284	11.62%	22264.70
stable matching_ks8_s8	93320	[10]	19	109836	-17.70%	102.66	3	105890	-13.46%	35663.77
geometric mean	**780.06**			**781.54**	**-1.90%**			**769.87**	**1.31%**	

by *garbling gadget*, we explore the possibility of using other symmetric logic gates, instead of AND2 gates. Based on a thorough study of 3-input symmetric logic gates, we make two proposals to reduce garbling costs: (1) merging AND2s in XAGs into wider ANDs, and (2) using X1G as the logic representation. The two proposals present two technical orientations to solve this problem and each of them stresses different aspects: the former strikes a good balance between gain and runtime cost, while the latter emphasizes particularly gain. Extensive evaluation of various benchmark suites demonstrates that our proposals have achieved significant improvement: compared to best-known results, our elaborated XAG- and X1G-targeted flows have respectively reduced the garbling costs by up to 25.27% and 35.48%; For the random/control circuits in the EPFL benchmark suite, the flows have achieved improvements of 7.66% and 19.92% on average.

ACKNOWLEDGMENT

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

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] R. Gilad-Bachrach, N. Dowlin, K. Laine *et al.*, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International conference on machine learning*. PMLR, 2016, pp. 201–210.

[6] P. Mishra, R. Lehmkuhl, A. Srinivasan *et al.*, "Delphi: A cryptographic inference service for neural networks," in *29th USENIX Security Symposium*, 2020, pp. 2505–2522.

[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] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi *et al.*, "Tinygarble: Highly compressed and scalable sequential garbled circuits," in *IEEE Symposium on Security and Privacy*, 2015, pp. 411–428.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] J. Boyar, R. Peralta, and D. Pochuev, "On the multiplicative complexity of boolean functions over the basis ($\wedge$, $\oplus$, 1)," *Theoretical Computer Science*, vol. 235, no. 1, pp. 43–57, 2000.

[15] 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.

[16] G. Meuli, M. Soeken, E. Campbell *et al.*, "The role of multiplicative complexity in compiling low t-count oracle circuits," in *IEEE/ACM International Conference on Computer-Aided Design*, 2019, pp. 1–8.

[17] J. Boyar and R. Peralta, "Tight bounds for the multiplicative complexity of symmetric functions," *Theoretical Computer Science*, vol. 396, no. 1-3, pp. 223–246, 2008.

[18] T. Häner and M. Soeken, "The multiplicative complexity of interval checking," *arXiv preprint arXiv:2201.10200*, 2022.

[19] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient fpga mapping solution," in *Proceedings of the ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999, pp. 29–35.

[20] H. Riener, W. Haaswijk, A. Mishchenko *et al.*, "On-the-fly and dag-aware: Rewriting boolean networks with exact synthesis," in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2019, pp. 1649–1654.

[21] Z. Huang, L. Wang, Y. Nasikovskiy *et al.*, "Fast boolean matching based on npn classification," in *International Conference on Field-Programmable Technology*. IEEE, 2013, pp. 310–313.

[22] D. S. Marakkalage, E. Testa, H. Riener *et al.*, "Three-input gates for logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 10, pp. 2184–2188, 2020.

[23] 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.

[24] E. L. Lawler, "An approach to multilevel boolean minimization," *Journal of the ACM*, vol. 11, no. 3, pp. 283–295, 1964.

[25] 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.

[26] N. Eén and N. Sorensson, "An extensible sat-solver," *Lecture notes in computer science*, vol. 2919, no. 2004, pp. 502–518, 2004.

[27] M. Soeken, H. Riener, W. Haaswijk *et al.*, "The epfl logic synthesis libraries," *arXiv preprint arXiv:1805.05121*, 2018.

[28] 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.

[29] 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.