cryptography

MDPI

Article

Garbled Circuits Reimagined: Logic Synthesis Unleashes Efficient Secure Computation

Mingfei Yu *, Dewmini Sudara Marakkalage and Giovanni De Micheli *

Integrated System Laboratory, École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland; dewmini.marakkalage@epfl.ch

* Correspondence: mingfei.yu@epfl.ch (M.Y.); giovanni.demicheli@epfl.ch (G.D.M.)

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 for 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 minimal structural multiplicative complexity (MC). Starting with a thorough study of the cipher-text efficiency of different types of logic primitives, for the first time, we propose XOR-OneHot graph (X1G) as a suitable logic representation for the generation of low-cost GCs. Our contribution includes (a) an exact algorithm to synthesize garbling-cost-optimal X1G implementations for small-scale functions and (b) a set of logic optimization algorithms customized for X1Gs, which together form a robust optimization flow that delivers high-quality X1Gs for practical functions. The effectiveness of the proposals is evidenced by comprehensive evaluations: compared with the state of the art, 7.34%, 26.14%, 13.51%, and 4.34% reductions in garbling costs are achieved on average for the involved benchmark suites, respectively, with reasonable runtime overheads.

Keywords: garbled circuits; logic synthesis; secure multiparty computation

Citation: Yu, M.; Marakkalage, D.S.; De Micheli, G. Garbled Circuits Reimagined: Logic Synthesis Unleashes Efficient Secure Computation. *Cryptography* **2023**, *7*, 61. https://doi.org/10.3390/cryptography7040061

Academic Editor: Jim Plusquellic

Received: 21 October 2023
Revised: 17 November 2023
Accepted: 21 November 2023
Published: 23 November 2023

1. Introduction

Secure computation, also known as secure multiparty 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 transactionsSt. Alban-Anlage 66, CH-4052 Basel, Switzerland [1] and voting systems [2].

Besides the well-known scenarios mentioned above, there is an increasing demand for privacy-preserving applications. For instance, in secure neural network inference, model providers avoid revealing their meticulously trained network models, and the clients avoid leaking their sensitive input data [3]. An increased interest in secure medical data sharing is being witnessed as well [4].

However, realizing general-purpose MPC is not easy. Among the few candidates in the literature, garbled circuit (GC) is a promising choice. First proposed by Andrew Yao in 1986 [5], GC is a cryptographic technique with a constant round complexity. GC offers several advantages over other competitors: compared with fully homomorphic encryption (FHE) [6], GC is free from prohibitive computational overhead and can be executed much faster; in comparison with secret sharing [7], GC protects the privacy of intermediate values, preventing information about private inputs from being revealed by intermediate results.

Meanwhile, this powerful technique also has its limitations. In this protocol, the target computation is represented by a Boolean circuit and subsequently transformed into

a GC—essentially a network of encrypted tables—for execution. The requirement that the encrypted tables shall be transmitted among parties results in an excessive amount of data communication. Considering that modern application scenarios are typically large-scale and usually have low latency and high energy efficiency requirements, 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 proposing better garbling schemes, because a smarter way to garble each logic gate can reduce the number of cipher texts in each encrypted table. To name a few, *Free-XOR* 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 [8]. *Half-gate* shows that any two-input nonlinear logic operation, such as AND2, can be garbled using two cipher texts, i.e., a two-entry encrypted table [9]. *Garbling gadget* proposes that the garbling cost of any m-input symmetric logic gate is no more than m, suggesting that large-fanin-sized logic gates might be more garbling-cost-efficient nonlinearity providers than AND2 [10].

The other technical direction to which this work belongs is to optimally synthesize the Boolean circuit that implements the target computation. Since XOR gates and inverters are free of garbling cost (by adopting the free-XOR scheme), this is widely regarded as a *multiplicative complexity* (MC) *reduction* problem: implement the target Boolean function over the basis of {AND, XOR, NOT}, i.e., as an *XOR-AND graph* (XAG), then minimize the number of two-input ANDs (AND2s) in the XAG through manipulating logic. Great success has been achieved: 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 [11,12]; Testa et al. create a logic synthesis toolbox, which leverages existing logic optimization techniques in a hybrid manner [13]; Liu et al. focuses on detecting the ANDs in a network that can be replaced by XNORs without changing the network's functionality [14].

For the first time, we raise the following question: "Is there a kind of logic representation more cipher-text-efficient than XAGs?" Aiming at finding garbling-cost-efficient logic primitives, our exploration started with a systematic study of the properties of three-input symmetric logic gates. Noticing that OneHots, whose truth table is $^{\#}16$ (in this paper, truth tables are represented in hexadecimal as a bit-string by default, and the most significant bit is on the left-hand side), can provide nonlinearity with more cipher-text efficiency than AND2s, a new logic representation, *XOR-OneHot graph* (X1G), is proposed (this manuscript is an extension and summary of the authors' (Yu, M., De Micheli, G.) previous works [15,16]. This work provides more detailed explanations of our techniques and involves a newly proposed logic optimization flow for X1Gs, which consists of three X1G optimization algorithms proposed for the first time). To unleash the efficiency of OneHots, powerful and agile logic optimization algorithms are elaborated, which together make up a customized optimization flow for X1Gs. All our proposed Boolean techniques are developed on top of the most advanced garbling schemes, which distinguishes this work from any existing effort made by the logic synthesis community.

This paper is organized as follows. Sections 2 and 3 provide the preliminaries, respectively, for the aspects of cryptographic protocol and logic synthesis. In Section 4, the cipher-text efficiency of different logic primitives in providing nonlinearity is analyzed; as a result, X1G is proposed as an ideal logic representation for low-cost GC generation. Based on studying the properties of OneHot operations, a mapping algorithm that bridges existing research on MC reduction and the new proposal of adopting X1Gs are proposed in Section 5. In Section 6, we propose a novelly formulated exact synthesis algorithm that can agilely synthesize optimal X1Gs for small-scale Boolean functions. By exploiting the exact synthesis algorithm, along with the Boolean function classification technique, a database of garbling cost-optimal X1G implementations for six-variable Boolean functions is generated. In Section 7, several heuristic logic optimization algorithms tailored for X1Gs are dedicated, which form an optimization flow that offers high-quality X1G implementations for practical

functions. Experimental results on evaluating the performance of (a) the exact synthesis algorithm and (b) the logic optimization flow are presented in Section 8. In Section 9, we discuss the potential influence of the runtime overhead of logic synthesis on system performance, which highlights the prospect of X1G-optimization-based low-cost GC generation techniques. Section 10 concludes this paper.

2. Garbled Circuits

2.1. The GC Protocol

The concept of GC was first proposed by Yao as a solution to secure two-party computation (2PC) [5] and was later generalized to an MPC protocol [17]. We introduce its 2PC version for clarity, which is also the core of the MPC variant. Generally, we adopt the formalization provided in [18].

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 synthesis*. 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 $En(\cdot)$: $\mathbb{B} \rightarrow \mathbb{B}^l$ that maps the two potential binary values, 0 and 1, to two l-bit labels. For example, the label A that corresponds to Alice's private input a is created following $A = En(a)$. 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 $De(\cdot)$: $\mathbb{B}^l \rightarrow \mathbb{B}$, while Bob shares the evaluation result Y. These two resources of information jointly determine the computation result y, as $y = De(Y)$.

2.2. Bottleneck of System Performance

A GC is essentially a network of encrypted tables. While preserving privacy, the cipher texts in encrypted tables are also the source of communication costs. A correct execution of the GC protocol requires the distribution of all involved cipher texts among the participants. This data communication is typically prohibitive, which easily bottlenecks system performance and has significantly impeded the spread and application of GC-based secure computation. For instance, garbling a three-layer neural network model already results in a 128 MB data communication [19]. The amount of data communication required to execute the protocol is hereinafter referred to as the *garbling cost*.

Given a GC, its garbling cost is determined by the number of cipher texts (denoted by "#cipher-texts") utilized to garble it; The terms "garbling cost" and "#cipher-texts" are therefore used interchangeably in this paper. As introduced, a GC is constructed on top of a Boolean circuit that implements the target computation. The Boolean circuit and the generated GC are isomorphic, as each logic gate in the former corresponds to an encrypted table in the latter. This fact suggests two orthogonal solutions to generating lower-cost GCs:

(a) In the stage of Boolean circuit synthesis, the garbling cost of each kind of logic gate shall be taken into consideration, so as to prioritize the usage of gates that require fewer cipher texts to garble. (b) More advanced garbling schemes are demanded, which facilitate using fewer cipher texts (i.e., a fewer-entry encrypted table) to garble each kind or certain types of logic gates.

2.3. Advanced Garbling Schemes

Despite numerous sophisticatedly elaborated garbling schemes in the literature, we introduce two that have directly inspired this work.

2.3.1. Free-XOR

Free-XOR points out that XOR gates can be garbled without using any cipher text, regardless of the fanin size [8].

An XOR operation is essentially a mod-2 addition. Hence, the bitwise XOR-ed result of m l-bit bitstrings is also an l-bit bitstring. Let there be such an m-input XOR gate in the Boolean circuit to be garbled, $y = XOR(x_1, \ldots, x_m)$. An encoding function En is desired to map the binaries $x_1, \ldots, x_m$ and y, to the bitstrings $X_1, \ldots, X_m$ and Y, such that $Y = \bigoplus_{i=1}^{m} X_i$, where "$\bigoplus$" denotes bitwise XOR operations. If such an encoding function is available, it implies the garbling of an XOR gate does not require any cipher texts—indeed, the executor can obtain the output label of a garbled XOR gate by conducting bitwise XOR operation on the input labels, instead of relying on an encrypted table for look-up and decryption.

The desired encoding is proven to be achievable [8]: Let s be a signal in the Boolean circuit to be garbled, and En, respectively, encodes $s = 0$ and $s = 1$ to two l-bit labels S^0 and S^1. Let Δ be a l-bit bitstring and satisfy $\Delta = \bigoplus_{i=\{0,1\}} S^i$. Then, En is a qualified encoding function, as long as it holds the property that Δ is the same for all signal s in the Boolean circuit to be garbled.

2.3.2. Garbling Gadget

When synthesizing a Boolean circuit to be garbled, logic gates with more fanins are hardly considered, because the garbling cost of a logic gate increases exponentially as the size of its fanin increases. Naïvely garbling an m-input logic gate requires 2^m cipher texts, since the truth table of an m-input gate is 2^m-entry.

However, *garbling gadget* [10] has yielded a fresh perspective. Inspired by Free-XOR, garbling gadget suggests interpreting any symmetric logic operation as a modular addition. This is possible because the output of a symmetric logic gate depends exclusively on the Hamming weight of its input pattern. For an m-input symmetric gate, the Hamming weight of its input pattern ranges from 0 to m, determining that the minimum modulus z is no more than $m + 1$. Therefore, an m-input symmetric operation can be interpreted as a bitwise mod-z addition, followed by a projection gadget that encodes the sums back to binary. The garbling of a modular addition is free from cipher texts; A z-to-2 projection gadget, by further adopting a compatible garbling scheme, called *garbling row reduction* [20], can be garbled into a $(z - 1)$-entry encrypted table.

Garbling gadget reduces the garbling cost of an m-input symmetric logic gate from 2^m to no more than $m - 1$. The finding that larger-fanin-sized symmetric gates can be garbled efficiently makes it worthwhile to explore the possibility of using them as logic primitives to implement the Boolean circuits to be garbled.

3. Logic Synthesis

Logic synthesis generally refers to the process of converting a high-level description of a circuit into a lower-level representation, such as a gate-level netlist. It plays an important role in modern *electronic design automation* (EDA) flows, as it optimizes the designs of integrated circuits for a specified cost criterion. The goal is typically, but not limited to, optimizing area, delay, power consumption, etc.

In logic synthesis, a Boolean circuit is typically abstracted into a logic network for compact representation. A logic network, essentially a *directed acyclic graph* (DAG), allows efficient applications of graph-based optimization algorithms to manipulate designs.

3.1. Exact Synthesis

Exact Synthesis refers to the task of finding the provably optimum way to represent Boolean functions using allowed types of primitives. The definition of optimum depends on the cost criterion of the synthesis problem.

The exploration of how to efficiently solve an exact synthesis problem has a long history [21]. In recent years, thanks to the remarkable progress that has been achieved in developing performant *Boolean satisfiability* (SAT) solvers, formulating an exact synthesis problem as a SAT problem has become mainstream [22]. But even for the same exact synthesis problem, depending on the way it is formulated, the difficulty of solving it may vary a lot [23].

Exact synthesis problems are intractable, as determined by their intrinsic complexity [24]. Thus, the application of exact synthesis techniques is restricted to small-scale functions. While a high-quality formulation can mitigate the limitation to some extent, the weak scalability of exact synthesis techniques is insurmountable to overcome in principle.

3.2. Peephole Optimization

Most logic synthesis techniques are heuristic, because the target logic networks are usually of large scale and complex, making it infeasible to find the optimum solution in a reasonable time. A large portion of successful heuristics can be categorized as *peephole optimizations*. This refers to divide-and-conquer-like strategies, i.e., partitioning a network into subnetworks to break down the problem into a series of amenable ones. Such a partition commonly relies on the concept of *cuts* [25].

3.2.1. Cuts

A cut in a logic network is identified by its *root*, which is a node, and its *leaves*, which are a collection of nodes. A feasible set of leaves shall meet two properties: (a) there is at least one leaf on any path from a primary input to the root; (b) all the leaves are on at least one such path.

A cut is recognized as *k-feasible* if its number of leaves does not exceed k. Given a specified k, the process of finding all the k-feasible cuts in the target network is known as *cut enumeration* [26].

3.2.2. Logic Rewriting

Logic rewriting refers to the idea of optimizing a logic network by greedily replacing each part of the network, i.e., each cut, with the optimum, or optimal, implementation of the local function of this cut.

The optimization effect of a logic rewriting algorithm highly depends on the selection of k, as k determines the allowed number of leaves of each cut. A larger k allows the process to get closer to the global optimum—to take an extreme example, when k achieves the input size of the network, a logic rewriting algorithm then reduces to the exact synthesis problem of finding the optimum implementation for the function of each primary output. Certainly, due to scalability concerns, it is impossible to set k to be unreasonably large. Configuring k to four or five is common practice.

How the optimum implementations of each cut are obtained is another factor that distinguishes different variants of logic rewriting algorithms. Existing algorithms fall into the categories of finding the optimum implementation of each encountered cut by either (a) invoking an exact synthesis solver on the fly [15], or (b) referring to a prepared database of optimum implementations of small-scale Boolean functions. The intrinsic complexity of exact synthesis problems determines that (b) typically supports the selection of a larger k than (a) does.

Based on the functional completeness of the database, (b) can be further divided into two genres: the (i) *Boolean-mining*-based functionally incomplete one [27] and the (ii) functionally complete one [28]. Boolean mining requires that the Boolean functions that occur in practice be collected before entering the logic optimization procedure; Considering the privacy-preserving property of secure computation, such a premise is likely invalid. Thus, we recognize (i) as inapplicable to our case. In practice, (ii) is usually applied along with *Boolean function classification* techniques, since the number of logic functions increases double exponentially with input size.

3.3. Algebraic Rewriting

Algebraic optimization methods consider logic functions as objects of some algebra (not necessarily Boolean algebra) and consider algebra-specific manipulations to optimize them. While such abstractions disregard certain Boolean properties, this simplification leads to fast algorithms that scale well to large logic networks.

In algebraic rewriting, lightweight local transformations based on algebraic axioms are iteratively applied to reshape large logic networks, aiming to optimize a predefined cost function, such as the total area or delay. This technique is especially powerful in near-homogeneous logic networks, i.e., logic representations that allow few types of logic primitives. This is because such networks tend to be more structured, and hence, the eligibility of small portions of logic for algebraic transformations can be efficiently evaluated with less computational overhead.

3.4. Don't-Cares-Based Optimization

In logic synthesis, *don't care* (DC) refers to a condition for which the output of a logic function or circuit is not specified or does not matter. A DC condition typically arises from the interconnections of logic gates in a logic network, due to the existence of reconvergent paths [29]. According to the exact cause, DC conditions can be classified into *satisfiability don't cares* (SDCs) and *observability don't cares* (ODCs): The former refers to the input patterns that are never produced under any primary input assignments. The latter refers to the input patterns whose output, if flipped, would not make a difference to any primary output.

DCs bring flexibility into implementing Boolean functions. Therefore, DCs are often used to simplify the design and optimize the resulting logic circuit [14]. However, finding DCs typically suffers from complex computation. In the last few decades, a technical change from using *binary decision diagrams* (BDDs) to using SAT solvers as the tool to compute DC has been witnessed [30].

3.5. Boolean Function Classification

Boolean function classification is the process of categorizing Boolean functions into different classes based on various characteristics and properties of the functions. For example, since input negation, input permutation, and output negation do not change the combinational complexity of a Boolean function, the so-called *NPN classification* [31] is widely adopted in Boolean function analysis.

In some applications, such as cryptography, *spectral classification* provides a powerful tool to classify Boolean functions, because spectral operations preserve algebraic properties [32]. Based on an n-variable Boolean function $f(x_1, \cdots, x_i, \cdots, x_j, \cdots, x_n)$, the five spectral operations are as follows:

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

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

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

3.6. Multiplicative Complexity

Depending on the context, *multiplicative complexity* (MC) can refer to the feature of either a Boolean function or an XAG implementation of a function. To avoid ambiguity, we distinguish the two cases as *functional MC* and *structural MC*.

Given a Boolean function, its functional MC is the minimum number of AND2s sufficient to implement it over the basis {AND2, XOR2, NOT} (i.e., as an XAG) [33]. Functional MC commonly serves as an indicator of the nonlinearity of a function. Thus, finding the functional MC of an arbitrary Boolean function correlates with research fields other than cryptography as well. Though receiving wide attention, it is an intractable problem [34]. So far, the functional MC of any logic function with no more than 6 inputs is known [35]. Most ongoing research on functional MC focuses on Boolean functions that either have certain features, such as symmetry [36], or appear frequently in certain applications, such as the interval checking function [37].

By contrast, structural MC is a feature of an XAG: it refers to the number of AND2s in this logic network. Indeed, a Boolean function's functional MC lower bounds the structural MC of any XAG implementing this function. Regarding structural MC, a well-known logic synthesis problem, the MC reduction problem asks, "Given an XAG implementation of the target Boolean function (whose functional MC is typically unknown), how does one reduce the number of AND2s in it as much as possible by manipulating logic synthesis techniques?" Free-XOR connects the task of synthesizing practical GCs to the MC reduction problem—in an XAG, AND2 is the only type of logic primitive whose garbling requires cipher texts. Therefore, any progress in addressing the MC reduction problem contributes to the synthesis of lower-cost GCs.

4. A Cipher-Text-Efficient Logic Representation

According to the definition of functional MC introduced in Section 3.6, AND2 is an intuitive nonlinearity provider. However, this does not mean that AND2 provides nonlinearity in the most cipher-text-efficient manner. Holding a doubtful attitude towards its cipher-text efficiency, the exploration of this work starts with answering the question: "Is there any kind of logic primitives that can provide nonlinearity more cipher-text efficiently than AND2s do?"

With the awareness that symmetric logic gates can be much more efficiently garbled than previous expectations, as garbling gadget pointed out, it is especially worthwhile to investigate if such a candidate exists among symmetric logic gates.

4.1. MC Compactness

To quantify the efficiency of a logic gate in providing nonlinearity, we propose the concept of *MC compactness*. It is defined as the functional MC of a logic gate divided by the number of cipher texts required to garble this logic gate, e.g., the MC compactness of AND2 is $1/2 = 0.5$. To provide a given amount of nonlinearity, it is likely that fewer cipher texts are needed when a more MC-compact kind of logic gate is adopted as the primitive.

Symmetric gates with fanin sizes larger than three are out of the scope of this work, considering the inefficiency of applying logic synthesis techniques to logic networks consisting of large-fanin-sized primitives. No 3-input *asymmetric* logic gates are of interest, as the excessive garbling cost assures that none of them has a preferable MC compactness. The Hamming weight of a 3-bit pattern has four potential values (ranging from 0 to 3). Since, for a symmetric 3-input function, its output for each Hamming weight can be independently set, there are 16 (2^4) symmetric Boolean functions out of all 256 (2^8) 3-input Boolean functions. After canonicalizing the 14 nontrivial ones (i.e., excluding the two constant functions)

by applying input negation and output negation, there turn out to be 5 candidates: AND3 (In this paper, three-input ANDs and XORs are abbreviated as AND3s and XOR3s, with the fanin size explicitly indicated, so as to distinguish from their two-input counterparts. Fanin size might be omitted when a claim applies, regardless of fanin size. For other gate types, the names proposed in [38] are adopted), XOR3, Majority, OneHot, and Gamble. Furthermore, XOR3 is excluded, as its functional MC is zero and is not qualified as a valid nonlinearity provider.

Adopting garbling gadget as the scheme to garble a symmetric logic gate, the cost is determined by the modulus of the modular addition that the logic operation is interpreted into. An algorithm to calculate the minimum modulus z to interpret a symmetric logic function as a modular addition is given in the Appendix A (Algorithm A1). When z is determined, as introduced in Section 2.3.2, $z-1$ cipher texts are required to garble the logic gate. On the other hand, the functional MCs of all five candidates are known [35]. Thus, their MC compactness can be calculated.

As one can learn from Table 1, OneHot and Gamble share the feature that they can both be expressed as a mod-3 addition followed by a 3-to-2 project gadget, while other candidates (i.e., AND3 and Majority) require the modulus to be at least four. To see this for OneHot, denote the patterns whose Hamming weights are i as *HMW(i)*, then $HMW(0) = (0, 0, 0)$ and $HMW(3) = (1, 1, 1)$. Since

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

the output of OneHot depends only on the Hamming weight Modulo 3. The same reasoning applies to Gamble.

Table 1. Features of the four 3-input symmetric logic gates of interest, with garbling gadget adopted as the garbling scheme.

Gate Type	Truth Table	Functional MC	Garbling Cost (#Cipher-Texts)	MC Compactness
AND3	$^{\#}80$	2	3	0.67
Majority	$^{\#}e8$	1	3	0.23
OneHot	$^{\#}16$	2	2	1.00
Gamble	$^{\#}81$	1	2	0.50

In addition, the functional MC of OneHot is two [38]. Low garbling cost and high functional MC have jointly determined OneHot as a cipher-text-efficient logic primitive to provide nonlinearity. This observation indicates that OneHots, together with XORs and NOTs, serve as a cipher-text-efficient logic representation. This representation, which we hereinafter term *XOR-OneHot graph* (X1G), possibly offers a better representation for Boolean functions that are to be garbled.

4.2. A Study on the Properties of OneHot Gates

Noticing that a NOT gate can be regarded as an XOR2 gate with an input fixed as constant one, as

$$NOT(x) = XOR2(1, x),$$

OneHot is therefore the only kind of logic primitive in X1Gs whose garbling requires cipher texts. Thus, we define the *X1G optimization problem* as the logic optimization problem of reducing the number of OneHots in a given X1G.

The fact that the functional MC of a OneHot operation is two (also witnessed by its structural MC-optimum XAG implementation in Figure 1, where "$\wedge$" and "$\oplus$" denote AND and XOR, respectively) points out the major advantage of adopting OneHot as the nonlinearity provider. In principle, an AND3 operation equals two consecutive AND2 operations and corresponds to two units of nonlinearity. When garbling an AND3 operation,

by adopting AND2 or AND3 as the nonlinearity provider, 4 (2×2) or 3 cipher texts would be required, respectively; By contrast, by a simple algebraic manipulation on the *algebraic normal form* (ANF) [35] of the OneHot operation:

$$OneHot(x_1, x_2, x_3) = x_1x_2x_3 \oplus x_1 \oplus x_2 \oplus x_3,$$

we obtain

$$AND3(x_1, x_2, x_3) = XOR2(OneHot(x_1, x_2, x_3), XOR3(x_1, x_2, x_3)) \quad (1)$$

Equation (1) points out that an AND3 operation can be realized using one OneHot and two XORs (as shown in Figure 2, where "OH" denotes OneHot); hence, the garbling cost of the X1G-based solution is merely two cipher texts.

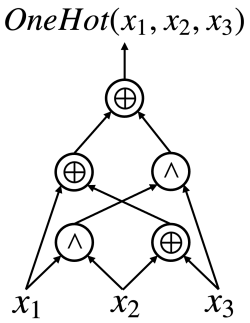

Figure 1. Structual MC-optimum XAG that implements OneHot operation.

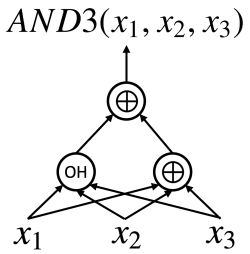

Figure 2. Implementing AND3 operation using OneHots and XORs.

By setting one of the three inputs of a OneHot gate to be a constant one, it is noticed that

$$OneHot(1, x_1, x_2) = AND2(\neg x_1, \neg x_2), \quad (2)$$

where "$\neg$" indicates negation.

Similarly, by configuring one input as constant zero, it is observed that

$$OneHot(0, x_1, x_2) = XOR2(x_1, x_2) \quad (3)$$

By applying this rewriting rule, from the left-hand side to the right-hand side, the number of OneHots in an X1G can be reduced. We, therefore, regard the application of this rule as a postprocessing stage in our logic optimization flow for X1Gs, which is introduced in the following sections.

5. Mapping XAGs to X1Gs Following Algebraic Rewriting Rules

To improve the practicality of the GC protocol, previous efforts made by the logic synthesis community focus exclusively on implementing the target function as an XAG and optimizing the network in the sense of reducing its structural MC. Thus, it is desired to elaborate an algorithm that maps an XAG to an X1G, which makes it possible to make full use of existing research and benefit from the cipher-text efficiency of OneHots at the same time.

5.1. An Algebraic-Rewriting-Based Mapping Approach

Equation (1) suggests replacing each pair of two adjacent AND2s with one OneHot and two XORs. Each application of this rule saves two cipher texts. But, it should be noted that this rewriting rule does not apply to any pair of adjacent AND2s: Given a pair of AND2s a and b, while a is a fanin of b, it is unwise to directly apply Equation (1) to get rid of a and b, as long as the fanout size of a is larger than one. This is because the logic function at node a is at the same time contributing to elsewhere in the logic network, and removing it would compromise the correct functionality of the network. Therefore, it is important to make sure that the algebraic rewriting is only applied to the pairs of AND2s without such concerns. To rule out this problem, we adopt the *covering* algorithm proposed in [39] as a preprocessing on the starting point XAG, which groups those AND2s that can be merged into one AND node with a larger fanin size.

For each AND2 that cannot be paired with another AND2, applying Equation (2) allows substituting each AND2 with a OneHot, with the amount of required cipher texts unchanged.

As described before, the *covering* function in line 1 in Algorithm 1 stands for the covering algorithm in [39]. Algorithm 1 allows mapping an already highly optimized XAG into an X1G of potentially further lower garbling cost. Furthermore, it has a linear time complexity and is algebraic-rewriting-based, which jointly guarantees an almost negligible runtime.

Algorithm 1: Mapping an XAG to an X1G by applying algebraic rewriting

Input: An XAG, N
Output: An X1G that is functionally equivalent to N

```
1  covering(N)
2  foreach AND node n ∈ N do
3      M ← decompose n into consecutive AND2s
4      while M ≠ ∅ do
5          if |M| = 1 then
6              {x1,x2}← fanins of M[0]
7              n'← OneHot(1, ¬x1, ¬x2) // Apply Equation (2)
8              replace node M[0] with n'
9              remove M[0] from M
10         else
11             {x1,x2}←fanins of M[0]
12             x3 ←the non-M[0] fanin of M[1]
13             n'← XOR2(OneHot(x1, x2, x3), XOR3(x1, x2, x3)) // Apply
                Equation (1)
14             replace nodes M[0] and M[1] with n'
15             remove M[0] and M[1] from M
16 return N
```

5.2. Limitations

Algorithm 1 serves as a runtime-friendly way to bridge existing research on MC reduction and the proposal of adopting OneHot as a cipher-text-efficient logic primitive. However, it is further noticed that relying on algebraic-rewriting-based mapping can direct us to suboptimality.

When judging the quality of an XAG implementation of the target function that is to be garbled, the number of AND2s in the network is the sole criterion, i.e., every AND2 is of equal cost in the context of MC reduction. This situation changes after OneHots are brought into the scope. When mapping an XAG into an X1G following Algorithm 1, depending on whether an AND2 can be paired with another, either Equation (1) or (2) would be applied to replace all AND2s using OneHots and XORs. However, Equation (1)

is the only rule that contributes to a lower garbling cost. In that sense, pairable AND2s, to which Equation (1) is applicable, are preferable to separated AND2s. This change in criterion explains the limitations of synthesizing X1Gs by applying Algorithm 1 to map the structural-MC-optimal XAGs optimized by previous works into X1Gs.

An example is given by the task of synthesizing the cipher-text-optimal X1G implementation for a 5-variable Boolean function, whose truth table is $^{\#}2888a000$ (when emphasizing the pursuit of fewer cipher texts, we regard the synthesized logic networks as *cipher-text/garbling-cost-optimal*, instead of *optimum*because, although we proved that OneHot serves as a more cipher-text-efficient logic primitive than AND2, its optimum remains an open question). It is one of the representatives of all 48 spectral-equivalent classes for 5-variable Boolean functions and has a known functional MC of three.

The optimality of the XAG implementation in Figure 3a is witnessed by the fact that its structural MC is three, equal to the functional MC of the logic function. Applying Algorithm 1 to it, however, cannot bring us the optimal X1G representation in Figure 3b—such an application would result in an X1G with three OneHots, whose garbling cost is six cipher texts, which is 50% worse than the optimal. Indeed, the expected XAG that can be mapped into the optimal X1G turns out to be the one with two pairs of adjacent AND2s in it. The structural MC of such an XAG is four, higher than the lower bound determined by the functional MC of the target function.

(**a**) (**b**)

Figure 3. Cipher-text-optimal implementations of function $^{\#}2888a000$: (**a**) Adopting XAG as the logic representation. (**b**) Adopting X1G as the logic representation.

That being said, to generate the cipher-text-optimal X1G implementation for a given function through applying the algebraic-rewriting-based mapping to its XAG implementation, the optimality of the starting point XAG should be taken care of by paying attention to (a) the number of AND2s and (b) their connectivities. A mapping starting from the XAGs that are optimized with structural MC adopted exclusively as the optimization goal in previous research can end up with suboptimality. Not only the amount but the environment of the AND2s also affects the quality of the X1Gs that the XAGs would be mapped into.

6. Agilely Synthesizing Optimal X1G Implementations for Small-Scale Functions

This section aims to formulate an exact synthesis problem of synthesizing the "truly" optimal XAG implementations for the given logic functions, which render optimal X1G implementations after applying Algorithm 1. The optimality is achieved by formulating both these two features regarding the usage of AND2s as the optimization objective of the exact synthesis.

Due to the limited scalability of exact synthesis, the approach is not applicable to Boolean functions of large scales for concerns of practicality. However, experimental evaluations witnessed that we managed to formulate the problem in such an efficient way that the optimal implementations are successfully synthesized for all 150,357 representative Boolean functions of 6-variable spectral-equivalent classes. These implementations of small functions can be used as building blocks for obtaining high-quality X1G implementations for practical functions.

6.1. AND Fence

We propose the concept of *AND fence* to effectively extract the information of our interest from a given XAG. Inspired by *Boolean fences* proposed in [23], which refers to the topology of a whole logic network, we propose AND fences to exclusively describe the usage of AND2s in XAGs.

The procedure for obtaining the AND fence consists of the following: (1) Apply the *covering* algorithm to classify AND2s in the network into groups. Each group of AND2s forms an AND tree. The cardinality of the i-th group is denoted as c_i, then $c_i \geq 1$. (2) The cardinalities of all d AND2 groups form the AND fence $\mathcal{F}$ of the network, i.e.,

$$\mathcal{F} = \{c_1, c_2, \cdots, c_d\} \tag{4}$$

The uniqueness of the AND fence of an XAG is ensured by ordering the AND2 groups by the network topology.

Intuitively, the structural MC of an XAG can be easily learned from its AND fence, as

$$smc(\mathcal{F}) = \sum_{i=1}^{d} c_i \tag{5}$$

More to the point, an XAG's AND fence contains all information required to calculate the garbling cost of the X1G that the XAG would map into, as

$$cost(\mathcal{F}) = \sum_{i=1}^{d} (2 \cdot \left\lceil \frac{c_i}{2} \right\rceil) \tag{6}$$

6.2. Abstract XAG

The formulated exact synthesis problem of synthesizing the cipher-text-optimal XAGs relies on a type of logic representation, named *abstract XAG*. In [35], the researchers simplified the network description from the original XAG to a more general form. First referred to as abstract XAG in [40], this logic representation distinguishes from XAG from the following perspectives: (1) Fanin sizes of XORs are allowed to be arbitrary (each XOR node is therefore called an *XOR cloud*). (2) Each fanin of any XOR cloud 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.

To make this representation compatible with the AND fence, we generalize it by allowing the fanin size of AND nodes to be more than two. Since a group of AND2s with a cardinality of c can be concisely represented as $(c+1)$-input AND, on top of the generalized abstract XAG, it is intuitive to both (a) extract the AND fence of a generalized abstract XAG and (b) construct a network topology of a generalized abstract XAG that meets a given AND fence. To avoid ambiguity, we hereinafter refer *abstract XAG* to our generalized representation.

Figure 4a provides an abstract XAG implementation of the Boolean function we saw before. The number of *steps* in an abstract XAG is defined as the number of ANDs in it. For conciseness, instead of using arrows, the blue texts under each XOR cloud denote the fanins of that XOR cloud. By focusing on the ANDs in the abstract XAG, the AND fence is easily extracted (Figure 4b), whose numerical representation is $\mathcal{F} = \{2, 2\}$, as both a_1 and a_2, the first-step and second-step ANDs, are three-input, indicating that the cardinalities of the two corresponding groups of AND2s are both two.

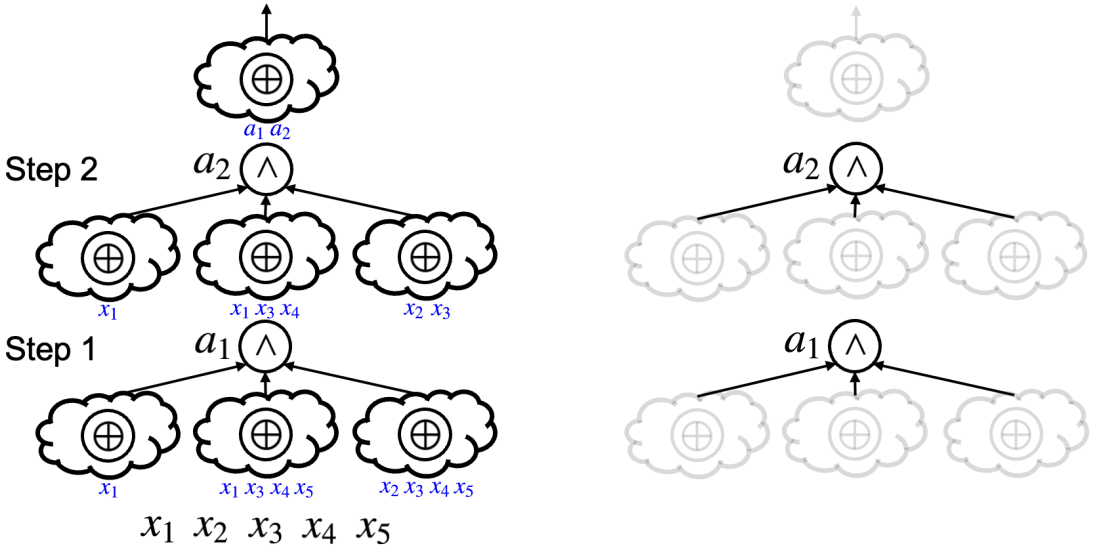

Figure 4. Abstract XAG and AND fence: (**a**) an abstract XAG implementation of Boolean function $^{\#}2888a000$ and (**b**) its AND fence.

6.3. Exactly Synthesizing Garbling-Cost-Optimal X1Gs

Utilizing AND fences and abstract XAGs, an exact algorithm to synthesize garbling-cost-optimal X1G implementations is elaborated.

6.3.1. Formulating an Exact Synthesis Problem

We formulate the exact synthesis problem of synthesizing the optimal XAG implementation for a given Boolean function as follows. Given a target AND fence, an incomplete abstract XAG is constructed, as the fanin connectivities of all XOR clouds are missing. A SAT solver is used to find if there is a way to configure the fanins of the XOR clouds so that the resulting abstract XAG implements the target function. The optimality of the synthesized abstract XAG is guaranteed by enumerating all candidates of AND fences in a garbling-cost-ascending manner. In other words, the formulated exact synthesis problem is composed of a series of instances, each of which explores the possibility of using a certain AND fence to implement the function.

The formulation relies on a library of AND fences. Indeed, the completeness of the library contributes directly to the optimality of the synthesized logic networks.

We start by addressing its simpler variant: "When targeting a certain structural MC, how many AND fences are there?" This is the reverse of calculating the structural MC of an AND fence (Equation (5)), and it is essentially a *positive integer partition* problem, a classic problem in number theory and combinatorics. Especially, the permutation of numerically different elements matters in our case, e.g., if $c_i \neq c_j$, $\{c_i, c_j\}$ and $\{c_j, c_i\}$ are two different AND fences. The algorithm proposed in [41] is adapted to our implementation.

It is known that the functional MC of up to six-variable Boolean functions is upper-bounded by six [35]. Therefore, by, respectively, figuring out the AND fence candidates with the structural MC ranging from one to six, all AND fences of interest are taken into consideration. There turns out to be 63 AND fence candidates in total in the library.

Algorithm 2 illustrates how the optimal abstract XAG implementation of a function is agilely synthesized. The synthesis procedure consists of solving a series of exact synthesis instances. Making use of a function's functional MC, the synthesis procedure is accelerated by skipping the instances where the involved AND fences are not able to provide sufficient structural MC (line 4).

The abstract XAG in Figure 4a is exactly the one found by the proposed approach, with $^{\#}2888a000$ given as the target Boolean function.

Algorithm 2: Agilely Synthesizing Optimal Abstract XAG using AND Fences

Input: Boolean function, f; Library of AND fences, *lib*
Output: Optimal abstract XAG implementation of f, N

```
fmc ← functional MC of f
foreach AND fence F ∈ lib do
    smc ← smc(F)
    if smc < fmc then continue
    N ← SAT(agile_formulation(f, F))
    if N ≠ NULL then return N
return N ← NULL
```

6.3.2. Effects of Suboptimality on XOR Counts

By decomposing the XOR clouds into XOR2s, an abstract XAG is converted into an XAG. How to conduct the decomposition is not trivial, as a naïve decomposition can result in functionally equivalent XOR2s, i.e., redundant logic in the resulting XAGs, as well as the X1Gs that the XAGs would map onto. Even if the decomposition is handled in an optimal manner, the resulting number of XOR2s is commonly beyond the minimal, as no efforts have been made to constrain the fanin size of the XOR clouds when synthesizing the optimal abstract XAG implementations.

While the pursuit of the minimal XOR2 count can be taken into consideration by introducing extra constraints to the exact synthesis problem, we recognize such an investigation as unwarranted for two reasons: (a) The garbling cost of an XAG does not depend on the number of XOR2s in it, as determined by the application of Free-XOR. (b) A significant advantage of the proposed exact synthesis problem formulation stems from the relaxation of XOR counts, which considerably increases the number of solutions in the search space and speeds up the problem-solving process.

By further applying Algorithm 1 to the XAGs that the synthesized abstract XAGs decompose into, the optimal X1G implementations are obtained. In this way, the limitation of directly applying Algorithm 1 to the structural-MC-optimal XAGs, which is introduced in Section 5.2, is overcome.

6.4. Database Generation

Despite the agility of the proposed formulation, due to the exponential complexity of exact synthesis problems, it is impractical to apply the method to exactly synthesize the X1G implementation for large-scale functions. To benefit from the formulation, we make use of it to build a functionally complete database, which contains the optimal X1G implementations of 6-input logic functions. The database later facilitates a database-driven logic rewriting algorithm, which plays a crucial role in our logic optimization flow for the X1G implementations of large-scale functions.

There are 2^{64} 6-input logic functions, and synthesizing the optimal X1G implementations for all of them is not a trivial task, even with the help of the agile formulation. By classifying Boolean functions, the number of functions to consider can be significantly reduced.

When choosing which classification approach to adopt, two factors are taken into consideration: (a) The functions classified into the same category should share the same cost. (b) When there is more than one option that meets (a), the one resulting in fewer classes is better. We recognize the spectral classification as a natural fit to our case.

As introduced in Section 3.5, for any two spectral-equivalent Boolean functions f and g, there exists a series of spectral 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 by adopting spectral classification is evidenced by the following theorem:

Theorem 1. *Assume the garbling-cost-optimal X1G implementation of function f is available, denoted as f_1. Then, by applying* **o** *to f_1, the obtained X1G g_1 is a garbling-cost-optimal X1G implementation of 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 OneHots than g_1. By applying $\mathbf{o}'$ to g_2, another implementation of function f, f_2, can be obtained. Since applying spectral-equivalent operations never changes the number of OneHots in an X1G, there are fewer OneHots 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. □

Thanks to the adoption of spectral classification, there are only 150,357 equivalent classes for all 6-input Boolean functions. For each class, a representative function is assigned following [42]. For the representative function of each class, the proposed approach is exploited to efficiently synthesize its optimal X1G implementation. In this way, a 150,357-entry database of the garbling-cost-optimal X1G implementations for 6-input Boolean functions is generated. The database facilitates a performant logic rewriting algorithm, which is introduced in the following section.

7. A Logic Optimization Flow for X1Gs

To synthesize X1G implementations for practical functions, a flow consisting of three logic optimization algorithms is elaborated (Figure 5).

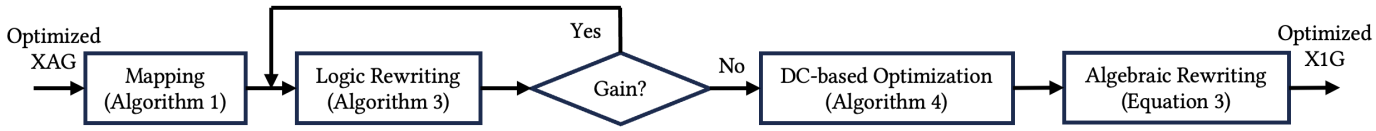

Figure 5. Proposed flow for X1G optimization.

The XAGs optimized by existing MC reduction techniques are adopted as the input to the flow. By applying algebraic-rewriting-based mapping (Algorithm 1), the XAGs are at first naïvely mapped onto X1Gs. Then, a database-driven logic rewriting algorithm is applied repetitively to the networks, until no gain in #OneHots reduction is observed. As postprocessing, which does not change the network topologies any more, a don't-care-based optimization approach and algebraic rewriting (Equation (3)) are exploited to further reduce the garbling cost.

7.1. Database-Driven Logic Rewriting

Algorithm 3 describes our database-driven logic rewriting algorithm.

For each node n in the network, a set of cuts rooted in it, denoted as **cuts**(n), are computed following the cut enumeration algorithm proposed in [26] (line 4). As soon as a part of the logic network is rewritten, cut enumeration shall be conducted dynamically.

For the logic cone highlighted by each cut $C \in$ **cuts**(n), using the prepared database, its optimal implementation can be constructed with negligible runtime overhead. Algorithm 3 is elaborated in such a greedy manner that the cut $C' \in$ **cuts**(n) that leads to the most significant reduction in local cost is to be committed (line 11, 12). Redundant nodes resulting from the rewritten operation are then removed (line 13).

If, for any node in the network, the cuts rooted in it are investigated, Algorithm 3 is terminated.

Algorithm 3: Database-driven logic rewriting

Input: X1G, N; Database of optimal X1Gs implementing small-scale functions, *db*
Output: Optimized X1G, N

1 *is_rewritten*$\leftarrow$ *true*
2 **foreach** node $n \in N$ in topological order **do**
3 **if** *is_rewritten* **then**
4 **cuts** $\leftarrow$ *cut_enumeration*(N)
5 *is_rewritten* $\leftarrow$ *false*
6 **foreach** cut $C \in$ **cuts**[n] **do**
7 **impl**$_{\textbf{old}}$[C] $\leftarrow$ the logic cone highlighted by cut C
8 **cost**$_{\textbf{old}}$[C] $\leftarrow$ cost of **impl**$_{\textbf{old}}$[C]
9 $f \leftarrow$local function of cut c
10 {**impl**$_{\textbf{new}}$[C], **cost**$_{\textbf{new}}$[C]} $\leftarrow$ *spectral_equivalent_classification*(f, *lib*)
11 cut $C' \leftarrow \arg\min_{C \in \textbf{cuts}[n]}$ (**cost**$_{\textbf{old}}$[C] - **cost**$_{\textbf{new}}$[C])
12 rewrite **impl**$_{\textbf{old}}$[C'] with **impl**$_{\textbf{new}}$[C']
13 $N \leftarrow$ *clean_up*(N)
14 *is_rewritten* $\leftarrow$ *true*
15 **return** N
16 **Function** *spectral_equivalent_classification*(f, *lib*) **:**
17 {representative r, operations **o** converting r to f} $\leftarrow$ *spectral_canonicalization*(f)
18 {optimal implementation of r $impl_r$, cost of $impl_r$ $cost_{new}$} $\leftarrow$ *lib*[r]
19 optimal implementation of f $impl_{new} \leftarrow$ apply **o** to $impl_r$
20 **return** {$impl_{new}$, $cost_{new}$}

7.2. Don't-Care-Based OneHot Reduction

The truth tables of a OneHot gate and an XOR3 gate are, respectively, $^{\#}$00010110 and $^{\#}$10010110 (represented in binary for intuitive comparison). In other words, the two types of gates are functionally distinguishable only by the input pattern $(1, 1, 1)$. This observation enlightens the possibility of exploiting DC conditions to replace some OneHots in an X1G with XOR3s, without affecting the functionality of the entire logic network. While both SDCs and ODCs can be utilized for logic optimization, we investigated the former in this work.

In Algorithm 4, the logic network is first converted into a logically equivalent *conjunctive normal form* (CNF) formula P, following the well-known Tseitin encoding [43] (line 1). For each OneHot node n, another CNF formula P' is constructed by adding extra clauses to P (line 5; "$\wedge$" denotes conjunction)—these clauses specify the input signals of node n to be constant ones. A SAT solver is then invoked to figure out if there is a set of variable assignments that satisfies P'. If not, it means that no primary input combination can produce pattern $(1, 1, 1)$ as the input of node n, indicating such a OneHot node is functionally indistinguishable from an XOR3 node with the same input. Hence, the logic network can be optimized by replacing n with an XOR3 node (line 7).

Algorithm 4: Don't-care-based logic optimization for X1G

Input: X1G, N
Output: Optimized X1G, N

1 CNF formula $P \leftarrow$ *Tseitin_encoding*(N)
2 **foreach** OneHot node $n \in N$ **do**
3 $\{x_1, x_2, x_3\} \leftarrow$ fanins of node n
4 $\{l_1, l_2, l_3\} \leftarrow$ literals in P that corresponds to signals $\{x_1, x_2, x_3\}$
5 CNF formula $P' \leftarrow P \wedge l_1 \wedge l_2 \wedge l_3$
6 **if** $SAT(P') \neq true$ **then**
7 replace n with an XOR3 node n', with the same fanins and fanouts as n
8 **return** N

8. Experimental Results

Here, we report the experimental evaluations of the effectiveness of (a) the exact synthesis formulation for agilely synthesizing cipher-text-optimal X1G implementations of small-scale functions, proposed in Section 6, and (b) the logic optimization flow for improving the X1G implementations of practical functions.

All experiments are conducted on an Apple M1 Max chip with 32 GB memory.

8.1. Evaluation on the Exact Synthesis Formulation

In [15], an exact synthesis algorithm is proposed to exactly synthesize optimal X1G implementation. The algorithm is at the core of the widely welcomed *single selection variable* (SSV)-based, area-oriented SAT encoding [22]. In addition, the heuristic of using functional MC to guide the targeted numbers of OneHots and XORs is integrated. We adopt this exact synthesis algorithm (hereinafter referred to as *baseline*) as the object of comparison in order to provide a convincing comparison.

In this experiment, the benchmark consists of all representative functions of the 48 spectral-equivalent classes for five-variable Boolean functions. The exact synthesis solver is implemented by exploiting the C++ reasoning library *bill* (available at https://github.com/lsils/bill), with *Glucose* (available at https://www.labri.fr/perso/lsimon/research/glucose/) adopted as the underneath SAT solver. The conflict limit for the SAT solver is set to 100,000.

As can be learned from Table 2, among the 48 target functions, the baseline algorithm failed to find any solution for five of them, while ours managed to synthesize X1G implementations for all of them. For seven functions, the solutions found by the baseline algorithm are suboptimal, as ours found solutions using fewer OneHots. This is evidenced by the total numbers of OneHots (column Accumulated #OneHots in Table 2) in the 43 solutions found, respectively, by the two approaches. These observations imply the infeasibility of relying on the baseline algorithm to generate a high-quality database for six-variable Boolean functions: while the algorithm is designed to be exact, in practice, even a reasonable conflict limit may divert it from the optimum.

Table 2. Synthesizing optimal X1G implementations for the 48 representatives of 5-variable spectral-equivalent classes.

Exact Synthesis Algorithm	#Solutions	Accumulated #OneHots	Accumulated Time [s]
Baseline	43	113	1354.58
Ours (Algorithm 2)	48	120 [1]	40.23

[1] This number goes down to 105 if we exclude the 5 cases where no solutions are synthesized following the baseline encoding.

In general, the X1G implementations synthesized following the proposed exact synthesis algorithm (Algorithm 2) always require fewer OneHots, together with an on-average 33.67× faster synthesis procedure. By capturing the significant characteristic of this practical GC generation problem in which XORs are free of garbling cost, Algorithm 2 smartly removes the constraint on XOR count in order to increase the number of solutions in the search space and make it possible for the SAT solver to find a solution efficiently.

8.2. Evaluation on the X1G Optimization Flow

In this experiment, three benchmark suites of different properties are involved to provide a comprehensive evaluation: the EPFL (available at https://github.com/lsils/benchmarks), the cryptographic (available at https://homes.esat.kuleuven.be/~nsmart/MPC/), and the MPC benchmark suites [12]. The EPFL benchmark suite consists of two kinds of combinational circuits, arithmetic ones and random/control ones. The crypto-

graphic benchmark suite involves block ciphers and other cryptographic functions. The MPC benchmark suite brings into scope popular MPC tasks, such as the secure auction and stable matching problems.

The garbling costs (i.e., #cipher-texts) of the X1Gs optimized by applying the proposed flow are compared with the state of the art, which is collected from three works in the literature [12–14], as none of them dominate the others in all of the benchmark suites involved. Since XAG is the adopted logic representation in these three works, the corresponding garbling costs are calculated as 2·#AND2s, where #AND2s denotes the number of AND2s in the structural-MC-optimal XAG implementations reported in the state of the art. These XAG implementations are adopted as the inputs to the proposed flow. As garbling a OneHot gate also requires two cipher texts (Table 1), the same cost as garbling an AND2, the garbling cost of an X1G is calculated as 2·#OneHots.

As for implementation details, all logic optimization algorithms involved in the proposed flow are implemented on top of the C++ logic network library *mockturtle* (available at https://github.com/lsils/mockturtle). In the implementation of the database-driven logic rewriting algorithm, we empirically set the maximum number of spectral operations allowed for matching (applying to the *spectral_canonicalization* function in line 17 of Algorithm 3) to be 100,000. For practical purposes, if the local function of a cut cannot be matched to any of the representatives within this limitation, such a cut is regarded as an unqualified candidate to operate rewriting. The SAT-based SDC computation in Algorithm 4 is implemented exploiting the C++ exact synthesis library *percy* (available at https://github.com/lsils/percy), with *MiniSAT* [44] adopted as the SAT solver. To avoid an excessive runtime cost in SDC computation, the solving of each SAT instance (line 6 of Algorithm 4) is bounded by a conflict limitation of 100,000. For the same reason, for benchmarks with more than 30,000 nodes, the DC-based optimization stage is bypassed.

In Table 3, we report the garbling costs after applying each stage of the optimization flow (#cipher-texts), the reductions in garbling costs achieved by applying the whole flow (red), and the overall runtime (Time). Due to the distinguishing features of the two kinds of benchmarks involved in the EPFL benchmark suite, we calculated the geometric means separately.

An interesting observation is that, by simply applying the mapping algorithm (Algorithm 1), a 2.41% reduction and an 11.90% reduction in garbling cost are achieved for the arithmetic circuits and random/control circuits, respectively. This finding evidences OneHot's superior cipher-text efficiency in providing nonlinearity compared with AND2. Although not included in the table due to space limitations, it is thrilling that this considerable reduction is obtained with an almost negligible runtime overhead. The most time-consuming case happens to the *log2* benchmark, which takes 0.48 s. Among all the random/control circuits, none of them require more than 0.10 s to finish.

Based on a more careful profile of the effectiveness of the logic optimization flow, logic rewriting (Algorithm 3) and mapping (Algorithm 1), respectively, contribute to 32.64% and 62.77% of the reduction in garbling costs, using 84.23% and 0.07% of the runtime. While the contribution made by DC-based optimization (Algorithm 4) and algebraic rewriting (Equation (3)) seems rather trivial, their roles in the flow are indeed crucial, as they managed to seize the optimization opportunities missed by the high-effort logic rewriting stage.

Through case studies on *Adder*, *Barrel shifter*, and *Decoder*, it is recognized that no reduction in #cipher-texts was achieved by the proposed flow because of benchmark features. Since the AND2s in the three starting point XAGs are separated, all OneHots in the three resulting X1Gs hold the feature described by Equation (2) and are unable to offer nonlinearity in a cipher-text-efficient manner. Therefore, the effectiveness of the proposed flow can vary from function to function, which is also learned from the evaluation results for the other two benchmark suites.

Observations from the experimental results in Tables 3 and 4 are consistent in general.

Table 3. Evaluating the X1G optimization flow for EPFL benchmark suite.

Benchmark	#Cipher-Texts	#Cipher-Texts (X1G Optimization Flow)				Red.	Time [s]
	(SOTA)	Mapping	Logic Rewrite	DC-Based Opt.	Algebraic Rewrite		
Adder	256	256	256 (1 *)	256	256	0.00%	1.72
Barrel shifter	1664	1664	1664 (1)	1664	1664	0.00%	1.88
Divider	10,264	9882	9294 (6)	9288	9288	9.51%	129.31
Log2	17,546	16,986	15,584 (4)	15,318	15,314	12.72%	532.73
Max	1744	1662	1636 (2)	1636	1636	6.19%	6.71
Multiplier	15,170	15,022	14,698 (3)	14,694	14,694	3.14%	116.10
Sine	3918	3724	3266 (5)	3244	3242	17.25%	214.06
Square-root	10,434	10,176	9406 (5)	9400	9400	9.91%	288.12
Square	9192	9052	8662 (4)	8648	8648	5.92%	58.63
Geometric mean	**4503.95**	**4395.44**	**4186.17**	**4173.57**	**4173.16**	**7.34%**	
Round-robin arbiter	2348	2260	1488 (2)	1488	1488	36.63%	62.23
Coding-cavlc	788	656	524 (2)	512	512	35.03%	39.57
ALU control unit	90	82	74 (3)	74	74	17.78%	2.87
Decoder	656	656	656 (1)	656	656	0.00%	1.93
i2c controller	1114	1004	886 (3)	872	872	21.72%	34.75
int to float converter	170	144	132 (3)	130	130	23.53%	9.15
Memory controller	9390	8298	7264 (5)	7156	7156	23.79%	296.11
Priority encoder	646	592	450 (2)	450	450	30.34%	16.06
Look-ahead XY router	186	126	114 (2)	114	114	38.71%	7.29
Voter	8514	7848	6344 (6)	6242	6242	26.69%	244.20
Geometric mean	**850.80**	**749.59**	**633.83**	**628.75**	**628.43**	**26.14%**	

* Data in parenthesis indicate the number of applications of the logic rewriting algorithm to reach saturation.

Table 4. Evaluating the X1G optimization flow for Cryptographic and MPC benchmark suites.

Benchmark	#Cipher-Texts	#Cipher-Texts (X1G Optimization Flow)				Red.	Time [s]
	(SOTA)	Mapping	Logic Rewrite	DC-Based Opt.	Algebraic Rewrite		
AES (Key Expansion)	10,880	10,240	10,240 (2)	10,240	10,240	5.88%	41.17
AES (No Key Expansion)	13,600	12,800	12,800 (2)	12,800 *	12,800	5.88%	50.22
DES (Key Expansion)	13,830	13,346	12,864 (4)	12,482	12,474	9.80%	533.58
DES (No Key Expansion)	13,666	13,194	12,690 (5)	12,280	12,278	10.16%	525.35
Comp. 32-bit SLT	168	138	120 (3)	120	120	28.57%	6.81
Comp. 32-bit SLTEQ	174	152	134 (2)	134	134	22.99%	7.24
Comp. 32-bit ULT	168	138	120 (3)	120	120	28.57%	6.81
Comp. 32-bit ULTEQ	174	152	134 (2)	134	134	22.99%	7.26
MD5	18,734	18,734	18,734 (1)	18,734	18,732	0.01%	27.61
SHA-1	22,966	22,834	22,636 (3)	22,636	22,636	1.44%	107.32
SHA-256	52,928	51,832	50,086 (3)	50,086	50,086	5.37%	335.79
Geometric mean	**3322.25**	**3066.13**	**2890.11**	**2873.61**	**2873.38**	**13.51%**	
Auction_2_16	194	194	194 (1)	194	194	0.00%	1.92
Auction_2_32	386	386	386 (1)	386	386	0.00%	1.92
Auction_3_16	464	464	460 (2)	460	460	0.86%	3.04
Auction_3_32	912	912	908 (2)	908	908	0.44%	3.11
Auction_4_16	990	990	986 (2)	986	986	0.40%	3.26
Auction_4_32	1950	1950	1946(2)	1946	1946	0.21%	3.44
Knn_comb_1_8	1108	1108	1100 (2)	1100	1100	0.72%	4.42
Knn_comb_1_16	2324	2324	2300 (2)	2300	2300	1.03%	4.87
Knn_comb_2_8	1762	1726	1676 (4)	1676	1676	4.88%	10.40
Knn_comb_2_16	3838	3754	3648 (4)	3648	3648	4.95%	15.21
Knn_comb_3_8	2120	2108	2082 (3)	2082	2082	1.79%	7.90
Knn_comb_3_16	4788	4760	4694 (3)	4694	4694	1.96%	13.64
Voting_1_3	14	14	12 (2)	12	12	14.29%	1.92
Voting_1_4	30	28	26 (2)	26	26	13.33%	2.18
Voting_2_2	42	40	38 (2)	38	38	9.52%	2.93
Voting_2_3	110	110	110 (1)	110	110	0.00%	2.37
Voting_2_4	208	208	204 (2)	204	204	1.92%	3.72
Voting_3_4	550	550	536 (2)	536	536	2.55%	7.47
Stable_matching_4_8	32,002	29,416	27,824 (6)	27,824	27,824	13.06%	379.88
Stable_matching_8_8	119,546	108,320	105,086 (3)	105,086	105,086	12.10%	502.04
Geometric mean	**790.95**	**777.04**	**757.74**	**756.64**	**756.64**	**4.34%**	

* Data in gray indicate that the DC-based optimization stage is skipped for the benchmark, since its size exceeds the predefined threshold.

It turns out that, for all functions in the MPC benchmark suite, the application of the algebraic rewriting rule did not lead to any reduction in garbling cost. Interestingly enough, for *MD5* in the cryptographic benchmark suite, the algebraic rewriting stage is the only one that achieved a lower garbling cost, even though it was by merely 0.01%. This fact demonstrates the unique role of each optimization approach involved in the proposed flow.

The least reduction in garbling cost is achieved for the MPC benchmark suite. This is likely explained by the following two facts: (a) an analysis of the benchmarks shows that AND2s are generally located far from each other; (b) with the experience of specialists, the benchmarks are designed to be garbling-cost-friendly, leaving little room for optimization.

9. Discussion

The experimental results in the previous section reveal the trade-off between the efforts in logic optimization and the runtime overhead of applying logic optimization algorithms. Considering this trade-off, X1G optimization algorithms/flows of different flavors are desired according to the target application scenario.

Optimally synthesizing Boolean circuits on the spot in an application-by-application manner can be time-consuming and may become a bottleneck of system performance. Under such circumstances, lightweight X1G optimization algorithms/flows that strike a balance between quality and speed are a suitable choice. According to our evaluation results, the mapping algorithm (Algorithm 1) and the algebraic rewriting rule (Equation (3)) provide such a solution.

Conversely, for frequent functions in secure computation, such as the MPC benchmark suite involved in the experimental evaluation, their X1G implementations can be optimally synthesized in advance and made publicly available. In this situation, high-effort logic algorithms, such as the logic rewriting algorithm (Algorithm 3) and the don't-care-based optimization algorithm (Algorithm 4), are preferable, since there is an unlimited time budget and exploiting any opportunities to reduce garbling cost is the priority. Notice that open-source optimized Boolean circuit implementations would not lead to any compromise in privacy, given that the security of the GC protocol comes from the encrypted tables that are created in the GC generation stage.

In future work, customized logic optimization algorithms supporting X1Gs can be designed in an application-scenario-aware manner, so as to meet the variant requirements of emerging MPC tasks.

10. Conclusions

Improving the efficiency of garbled circuits is more urgent than ever before to meet the rapidly increasing application requirements of MPC. Existing efforts made by the logic synthesis community focus intensively and exclusively on addressing the MC reduction problem, which is based on the premise of adopting XAGs as the logic representation for the target computation to be garbled. However, there is a lack of formal proof of the superiority of using XAGs as the underlying logic representation for the task of low-cost GC generation. Based on a thorough study of cipher texts' efficiency for logic gates, we propose here, for the first time, a more efficient logic representation based on X1Gs for generating efficient GCs. To support this claim, we showed (a) a novel exact synthesis algorithm to agilely synthesize the garbling-cost-optimal X1G implementations for small-scale functions and (b) a logic optimization flow for X1Gs, scalable to practical functions, that consists of a series of customized X1G optimization algorithms and that achieves high-quality X1G implementations. Comprehensive experimental evaluations for public benchmark suites evidenced the effectiveness of our proposals: compared with the state of the art, for the EPFL arithmetic, the EPFL random/control, the cryptographic, and the MPC benchmark suites, adopting X1Gs as the logic representation and applying our elaborated logic optimization algorithms jointly led to 7.34%, 26.14%, 13.51%, and 4.34% reductions, respectively, in garbling costs, with reasonable runtime overheads. This work offers a new

perspective for low-cost GC generation and enables practical high-performance secure computation.

Author Contributions: Conceptualization, M.Y. and G.D.M.; methodology, M.Y. and D.S.M.; software, M.Y.; validation, M.Y. and D.S.M.; formal analysis, M.Y.; investigation, M.Y.; resources, G.D.M.; data curation, M.Y.; writing—original draft preparation, M.Y. and D.S.M.; writing—review and editing, G.D.M.; visualization, M.Y.; supervision, G.D.M.; project administration, G.D.M.; funding acquisition, G.D.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by Synopsys Inc. and the SNF grant 200021_1920981.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All proposed algorithms and the generated database are available at https://github.com/MingfeiYu/mockturtle.

Acknowledgments: The authors would like to thank H.L. Liu for sharing the benchmarks optimized using Boolean techniques proposed in [14].

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

2PC	Secure two-party computation
ANF	Algebraic normal form
BDD	Binary decision diagram
DC	Don't-care
FHE	Fully homomorphic encryption
GC	Garbled circuits
MC	Multiplicative complexity
MPC	Secure multiparty computation
ODC	Observability don't-care
OT	Oblivious transfer
SAT	Boolean satisfiability
SDC	Satisfiability don't-care
SSV	Single selection variable
X1G	XOR-OneHot graph
XAG	XOR-AND graph

Appendix A

To figure out the minimum modulus z that is sufficient to express an m-input symmetric Boolean function as a modular addition, a simple algorithm is conceived based on the definition of modular additions.

Algorithm A1: Determining the minimum modulus to interpret a symmetric logic function as a modular addition

Input: m-input symmetric logic function, f
Output: Modulus to express the function as a modular addition, z

```
1 for i ← 2 to m do
2     for j ← i to m do
3         if f(HMW(j)) ≠ f(HMW(j mod i)) then
4             break
5         if j = m then
6             return z←i
7 return z ← m+1
```

In line 3 of Algorithm A1, *HMW(i)* denotes the input patterns whose Hamming weights are i. For example, in the cases of three-input Boolean functions ($m = 3$), *HMW(1)* refers to the following three input patterns: 001, 010, and 100.

References

1. Bogetoft, P.; Damgård, I.; Jakobsen, T.; Nielsen, K.; Pagter, J.; Toft, T. A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In Proceedings of the International Conference on Financial Cryptography and Data Security, Anguilla, British West Indies, 27 February–2 March 2006; pp. 142–147. [CrossRef]
2. Clarkson, M.R.; Chong, S.; Myers, A.C. Civitas: Toward a Secure Voting System. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, USA, 18–22 May 2008; pp. 354–368. [CrossRef]
3. Gilad-Bachrach, R.; Dowlin, N.; Laine, K.; Lauter, K.; Naehrig, M.; Wernsing, J. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In Proceedings of the 33rd International Conference on International Conference on Machine Learning, New York, NY, USA, 19–24 June 2016; Volume 48, pp. 201–210. [CrossRef]
4. Chen, F.; Jiang, X.; Wang, S.; Schilling, L.M.; Meeker, D.; Ong, T.; Matheny, M.E.; Doctor, J.N.; Ohno-Machado, L.; Vaidya, J. Perfectly Secure and Efficient Two-Party Electronic-Health-Record Linkage. *IEEE Internet Comput.* **2018**, *22*, 32–41. [CrossRef] [PubMed]
5. Yao, A.C.C. How to Generate and Exchange Secrets. In Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS 1986), Toronto, ON, Canada, 27–29 October 1986; pp. 162–167. [CrossRef]
6. Gentry, C. *A Fully Homomorphic Encryption Scheme*; Stanford University: Stanford, CA, USA, 2009.
7. Shamir, A. How to Share a Secret. *Commun. ACM* **1979**, *22*, 612–613. [CrossRef]
8. Kolesnikov, V.; Schneider, T. Improved garbled circuit: Free XOR gates and applications. In Proceedings of the International Colloquium on Automata, Languages, and Programming, Reykjavik, Iceland, 7–11 July 2008; pp. 486–498. [CrossRef]
9. Zahur, S.; Rosulek, M.; Evans, D. Two Halves Make a Whole. In Proceedings of the Advances in Cryptology—EUROCRYPT 2015, Sofia, Bulgaria, 26–30 April 2015; pp. 220–250. [CrossRef]
10. Ball, M.; Malkin, T.; Rosulek, M. Garbling Gadgets for Boolean and Arithmetic Circuits. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 565–577. [CrossRef]
11. Songhori, E.M.; Hussain, S.U.; Sadeghi, A.R.; Schneider, T.; Koushanfar, F. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In Proceedings of the IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 411–428. [CrossRef]
12. Riazi, M.S.; Javaheripi, M.; Hussain, S.U.; Koushanfar, F. MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation. In Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 5–10 May 2019; pp. 198–207. [CrossRef]
13. Testa, E.; Soeken, M.; Riener, H.; Amarù, L.; De Micheli, G. A Logic Synthesis Toolbox for Reducing the Multiplicative Complexity in Logic Networks. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 9–13 March 2020; pp. 568–573. [CrossRef]
14. Liu, H.L.; Li, Y.T.; Chen, Y.C.; Wang, C.Y. A Don't-Care-Based Approach to Reducing the Multiplicative Complexity in Logic Networks. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2022**, *41*, 4821–4825. [CrossRef]
15. Yu, M.; De Micheli, G. Generating Lower-Cost Garbled Circuits: Logic Synthesis Can Help. In Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), San Jose, CA, USA, 1–4 May 2023; pp. 304–314. [CrossRef]
16. Yu, M.; De Micheli, G. Striving for Both Quality and Speed: Logic Synthesis for Practical Garbled Circuits. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Francisco, CA, USA, 29 October–2 November 2023; accepted.
17. Beaver, D.; Micali, S.; Rogaway, P. The Round Complexity of Secure Protocols. In Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, 13–17 May 1990; pp. 503–513. [CrossRef]
18. Bellare, M.; Hoang, V.T.; Rogaway, P. Foundations of Garbled Circuits. In Proceedings of the ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; pp. 784–796. [CrossRef]
19. Ball, M.; Carmer, B.; Malkin, T.; Rosulek, M.; Schimanski, N. *Garbled Neural Networks Are Practical*; Cryptology ePrint Archive: 2019. Available online: https://eprint.iacr.org/2019/338 (accessed on 20 November 2023).
20. Naor, M.; Pinkas, B.; Sumner, R. Privacy Preserving Auctions and Mechanism Design. In Proceedings of the ACM Conference on Electronic Commerce, Denver, CO, USA, 3–5 November 1999; pp. 129–139. [CrossRef]
21. Lawler, E.L. An Approach to Multilevel Boolean Minimization. *J. ACM* **1964**, *11*, 283–295. [CrossRef]
22. Soeken, M.; Haaswijk, W.; Testa, E.; Mishchenko, A.; Amarù, L.; Brayton, R.K.; De Micheli, G. Practical exact synthesis. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 309–314. [CrossRef]
23. Haaswijk, W.; Soeken, M.; Mishchenko, A.; De Micheli, G. SAT-Based Exact Synthesis: Encodings, Topology Families, and Parallelism. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 871–884. [CrossRef]
24. Murray, C.D.; Williams, R.R. On the (Non) NP-Hardness of Computing Circuit Complexity. In Proceedings of the 30th Conference on Computational Complexity, Portland, OR, USA, 17–19 June 2015; pp. 365–380. [CrossRef]

25. Cong, J.; Wu, C.; Ding, Y. 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, Monterey, CA, USA, 21–23 February 1999; pp. 29–35. [CrossRef]
26. Mishchenko, A.; Chatterjee, S.; Brayton, R.K. Improvements to Technology Mapping for LUT-Based FPGAs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2007**, *26*, 240–253. [CrossRef]
27. Yang, W.; Wang, L.; Mishchenko, A. Lazy Man's Logic Synthesis. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 5–8 November 2012; pp. 597–604. [CrossRef]
28. Mishchenko, A.; Chatterjee, S.; Brayton, R. DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis. In Proceedings of the 43rd Annual Design Automation Conference, San Francisco, CA, USA, 24–28 July 2006; pp. 532–535. [CrossRef]
29. Brand. Redundancy and Don't Cares in Logic Synthesis. *IEEE Trans. Comput.* **1983**, *C-32*, 947–952. [CrossRef]
30. Mishchenko, A.; Brayton, R.K. SAT-Based Complete Don't-Care Computation for Network Optimization. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Munich, Germany, 7–11 March 2005; pp. 412–417. [CrossRef]
31. Goto, E.; Takahasi, T. Some Theorems Useful in Threshold Logic for Enumerating Boolean Functions. In Proceedings of the International Federation for Information Processing Congress, Munich, Germany, 27 August–1 September 1962; pp. 747–752.
32. Edwards, C. The Application of the Rademacher–Walsh Transform to Boolean Function Classification and Threshold Logic Synthesis. *IEEE Trans. Comput.* **1975**, *C-24*, 48–62. [CrossRef]
33. Boyar, J.; Peralta, R.; Pochuev, D. On the Multiplicative Complexity of Boolean Functions over the Basis ($\wedge$, $\oplus$, 1). *Theor. Comput. Sci.* **2000**, *235*, 43–57. [CrossRef]
34. Sönmez Turan, M. *New Bounds on the Multiplicative Complexity of Boolean Functions*; Cryptology ePrint Archive: 2022. Available online: https://eprint.iacr.org/2022/1077 (accessed on 20 November 2023).
35. Çalık, Ç.; Sönmez Turan, M.; Peralta, R. The Multiplicative Complexity of 6-variable Boolean Functions. *Cryptogr. Commun.* **2019**, *11*, 93–107. [CrossRef]
36. Brandão, L.T.A.N.; Çalık, C.; Sönmez Turan, M.; Peralta, R. Upper Bounds on the Multiplicative Complexity of Symmetric Boolean Functions. *Cryptogr. Commun.* **2019**, *11*, 1339–1362. [CrossRef]
37. Häner, T.; Soeken, M. The multiplicative complexity of interval checking. *arXiv* **2022**, arXiv:2201.10200.
38. Marakkalage, D.S.; Testa, E.; Riener, H.; Mishchenko, A.; Soeken, M.; De Micheli, G. Three-Input Gates for Logic Synthesis. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *40*, 2184–2188. [CrossRef]
39. Mishchenko, A.; Brayton, R.; Jang, S.; Kravets, V. Delay Optimization using SOP Balancing. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 7–10 November 2011; pp. 375–382. [CrossRef]
40. Soeken, M. Determining the Multiplicative Complexity of Boolean Functions using SAT. *arXiv* **2020**, arXiv:2005.01778.
41. Knuth, D.E. *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees–History of Combinatorial Generation*; Addison-Wesley Professional: Sebastopol, CA, USA, 2013.
42. Miller, D.M.; Soeken, M. An Algorithm for Linear, Affine and Spectral Classification of Boolean Functions. In *Advanced Boolean Techniques*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 195–215. [CrossRef]
43. Tseitin, G.S. On the Complexity of Derivation in Propositional Calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*; Springer: Berlin/Heidelberg, Germany, 1983; pp. 466–483. [CrossRef]
44. Eén, N.; Sörensson, N. An Extensible SAT-solver. In Proceedings of the Theory and Applications of Satisfiability Testing, Barcelona, Spain, 5–9 July 2004; Springer: Berlin/Heidelberg, Germany, 2004; pp. 502–518. [CrossRef]