Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting

DongKwon Lee
Seoul National University
Korea
dklee@ropas.snu.ac.kr

Woosuk Lee*
Hanyang University
Korea
woosuk@hanyang.ac.kr

Hakjoo Oh
Korea University
Korea
hakjoo_oh@korea.ac.kr

Kwangkeun Yi
Seoul National University
Korea
kwang@ropas.snu.ac.kr

Abstract

We present a new and general method for optimizing homomorphic evaluation circuits. Although fully homomorphic encryption (FHE) holds the promise of enabling safe and secure third party computation, building FHE applications has been challenging due to their high computational costs. Domain-specific optimizations require a great deal of expertise on the underlying FHE schemes, and FHE compilers that aims to lower the hurdle, generate outcomes that are typically sub-optimal as they rely on manually-developed optimization rules. In this paper, based on the prior work of FHE compilers, we propose a method for automatically learning and using optimization rules for FHE circuits. Our method focuses on reducing the maximum multiplicative depth, the decisive performance bottleneck, of FHE circuits by combining program synthesis and term rewriting. It first uses program synthesis to learn equivalences of small circuits as rewrite rules from a set of training circuits. Then, we perform term rewriting on the input circuit to obtain a new circuit that has lower multiplicative depth. Our rewriting method maximally generalizes the learned rules based on the equational matching and its soundness and termination properties are formally proven. Experimental results show that our method generates circuits that can be homomorphically evaluated 1.18x – 3.71x faster (with the geometric mean of 2.05x) than the state-of-the-art method. Our method is also orthogonal to existing domain-specific optimizations.

*Corresponding author.

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385996

CCS Concepts: • **Theory of computation → Circuit complexity**; **Cryptographic primitives**; • **Software and its engineering → Search-based software engineering**.

Keywords: Homomorphic Encryption Circuit, Program Synthesis, Term Rewriting

ACM Reference Format:
DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3385996

1 Introduction

Fully Homomorphic Encryption (FHE) [30] enables safe and secure third-party computation with private data because it enables any computation on encrypted data without the decryption key. Because the cloud can perform any computation on encrypted data without learning anything about the data itself, the clients can safely upload their encrypted data without any need to trust the software/hardware vendors of the cloud.

However, building FHE applications has been challenging at the moment because of their high computational costs. Though building FHE applications itself does not require much expertise thanks to off-the-shelf libraries of FHE schemes [31, 32, 44], when naively implemented, even with the best FHE schemes [10, 18], FHE applications incur slowdown factors of orders of magnitudes compared to their plaintext version. One of the key challenges is therefore reducing the costs of FHE applications amenable to practical use.

Existing Approaches. There have been two approaches – domain-specific optimizations and optimizing FHE compilers – for reducing the costs of FHE applications, but they are still less powerful than desirable. Various domain-specific FHE optimization techniques have been successfully developed, but developing such techniques requires a great deal of

expertise on the underlying FHE schemes. For example, optimizations such as rescaling [23], data movement [38] and batching [39], and heuristics for encryption parameter selection [23, 25] enable several orders of magnitude speedups in a wide range of FHE applications (such as image recognition [23], statistical analysis [38], sorting [17], bioinformatics [19], database [8], and static program analysis [37]), yet required a great deal of expertise in cryptography. Lowering this hurdle of expertise is a goal of FHE compilers, which, equipped with FHE optimization techniques, automatically transform conventional high-level programs into optimized FHE code. For example, Ramparts [4], Cingulata [15] and Alchemy [22] take programs written in a high-level language (e.g., Julia, C++, a custom DSL, resp.) and transform them into arithmetic circuit representations which can be evaluated using a backend FHE scheme. However, though the users do not have to concern about low-level details of underlying schemes when writing applications, they need to write *FHE-friendly* algorithms [16, 17, 19, 38] to achieve the desired efficiency, which still requires expertise. Furthermore, state-of-the-art compilers rely on hand-written optimization rules and devising such rules requires expertise and is likely to be sub-optimal.

Our Approach. In this paper, based on the prior work of FHE compilers, we propose a novel and general method for optimizing FHE boolean circuits that outperforms existing automatic FHE optimization techniques. Our method focuses on reducing the number of nested multiplications and achieves sizeable optimizations even for existing domain-specific manually optimized results.

Our setting of the optimization problem is simple. Let c be an arithmetic code that can be evaluated using FHE schemes, which can be either generated by a FHE compiler or written by a developer. Optimization is to find a new circuit c' of lower computational cost while guaranteeing the same semantics as the original. Because the decisive performance bottleneck in homomorphic computation is the nested depth of multiplications [4, 15, 17, 49], we set the computation cost to be measured using *the maximum multiplicative depth*, which is simply the maximum number of sequential multiplications required to perform the computation. For example, the circuit $c(x_1, x_2, x_3, x_4, x_5) = ((x_1x_2)x_3)x_4 + x_5$ has multiplicative depth 3. The lower the multiplicative depth is, the more efficiently a circuit can be evaluated. For example, we can optimize c by replacing it with $c'(x_1, x_2, x_3, x_4, x_5) = (x_1x_2)(x_3x_4) + x_5$ that has depth 2.

To achieve this critical optimization for homomorphic computation circuits as much as possible, we combine two techniques: program synthesis and term rewriting. Fig. 1 depicts our approach.

- Our method first automatically learns equivalences of small circuits from a set of training circuits using the program synthesis technique and then uses the learned equivalences to optimize unseen circuits. To learn such equivalences, we partition each training circuit into multiple sub-parts and synthesize their equivalent counterparts of smaller depth.
  These machine-found optimization patterns are so surprisingly aggressive that we can hardly imagine them from human-devised optimization techniques.
- Next, armored with these automatically learned equivalences as rewrite rules, we perform term rewriting on the input circuit to obtain a new circuit that has lower multiplicative depth. We maximally generalize what have been learned from training circuits by giving flexibility to the rewriting procedure: our method is based on the *equational matching* instead of the syntactic matching. Our rewriting method is proven to be sound and terminating.

We implement our method atop Cingulata [21], a widely used FHE compiler and evaluate our method on 18 FHE applications from diverse domains (statistical analysis, sorting, medical diagnosis, low-level algorithm). We learn rewrite rules from a set of Cingulata-generated Boolean circuits and apply the rules into other circuits.[1] On average, our method generates Boolean circuits that can be homomorphically evaluated 1.18x – 3.71x faster (with the geometric mean of 2.05x) than the state-of-the-art method [14].

Contributions.

- A novel general method for optimizing homomorphic evaluation circuits: we first learn rewrite rules from a set of training Boolean circuits using program synthesis and then perform term-rewriting on a given new circuit. The soundness and termination properties of this rewriting system are formally proven.
- Confirming the method's effectiveness in a realistic setting – the method outperforms existing automatic FHE optimization techniques, and even shows sizeable optimizations for domain-specific manually optimized results.

2 Problem Definition

We define the problem of minimizing the multiplicative depth of Boolean circuits. We first provide background on homomorphic encryption (Section 2.1) and Boolean circuits (Section 2.2). In Section 2.3, we formally state the problem.

2.1 Homomorphic Encryption

A fully homomorphic encryption (FHE) scheme with binary plaintext space $\mathbb{Z}_2 = \{0, 1\}$ can be described by the interface:

$$\mathsf{Enc}_{pk} : \mathbb{Z}_2 \to \Omega \qquad \mathsf{Dec}_{sk} : \Omega \to \mathbb{Z}_2$$
$$\mathsf{Add}_{pk} : \Omega \times \Omega \to \Omega \qquad \mathsf{Mul}_{pk} : \Omega \times \Omega \to \Omega$$

[1] Although the paper targets Boolean circuits, the method can also be directly applied to arithmetic circuits.

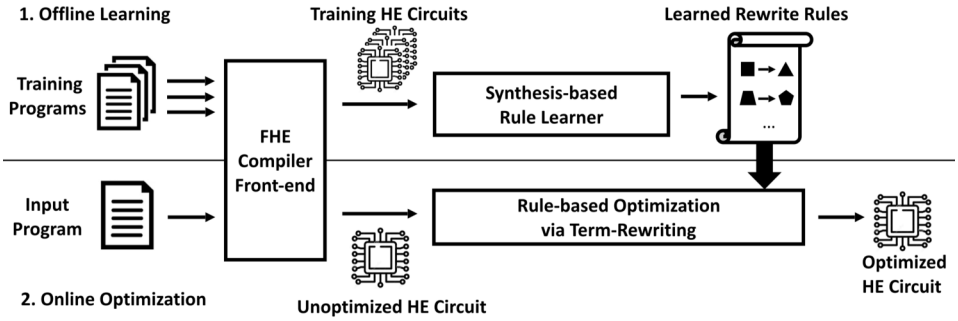

Figure 1. Overview of the system.

where pk is a public key, sk is a secret key, Ω is a ciphertext space (e.g. large cardinality set such as $\mathbb{Z}_q$, i.e., integers modulo an integer q). For all plaintexts $m_1, m_2 \in \mathbb{Z}_2$, the interface should satisfy the following algebraic properties:

$$\mathsf{Dec}_{sk}(\mathsf{Add}_{pk}(\mathsf{Enc}_{pk}(m_1), \mathsf{Enc}_{pk}(m_2))) = m_1 + m_2,$$
$$\mathsf{Dec}_{sk}(\mathsf{Mul}_{pk}(\mathsf{Enc}_{pk}(m_1), \mathsf{Enc}_{pk}(m_2))) = m_1 \times m_2.$$

Note that such a scheme is able to potentially evaluate all Boolean circuits as addition and multiplication in $\mathbb{Z}_2$ correspond to XOR and AND operations.

As an instance, let us consider a simple symmetric version (where only a secret key is used for both encryption and decryption) of the HE scheme [24] based on approximate common divisor problems [34]:

- The secret key (sk) is a random integer p.
- For a plaintext m, $\mathsf{Enc}(m)$ outputs $pq + 2r + m$, where q and r are randomly chosen integers such that $|r| \ll |p|$. r is a noise for ensuring semantic security [40].
- For a ciphertext $\bar{c}$, $\mathsf{Dec}(\bar{c})$ outputs $((\bar{c} \bmod p) \bmod 2)$.
- For ciphertexts $\bar{c}_1$ and $\bar{c}_2$, $\mathsf{Add}(\bar{c}_1, \bar{c}_2)$ outputs $\bar{c}_1 + \bar{c}_2$.
- For ciphertexts $\bar{c}_1$ and $\bar{c}_2$, $\mathsf{Mul}(\bar{c}_1, \bar{c}_2)$ outputs $\bar{c}_1 \times \bar{c}_2$.

For ciphertexts $\bar{c}_1 \leftarrow \mathsf{Enc}(m_1)$ and $\bar{c}_2 \leftarrow \mathsf{Enc}(m_2)$, we know each $\bar{c}_i$ is of the form $\bar{c}_i = pq_i + 2r_i + m_i$ for some integer q_i and noise r_i. Hence $\mathsf{Dec}(\bar{c}_i) = (\bar{c}_i \bmod p) \bmod 2) = m_i$, if $|2r_i + m_i| < p/2$. Then, the following equations hold:

$$\bar{c}_1 + \bar{c}_2 = p(q_1 + q_2) + \underbrace{2(r_1 + r_2) + m_1 + m_2}_{\mathrm{noise}_{\mathsf{Add}}}$$

$$\bar{c}_1 \times \bar{c}_2 = p(pq_1q_2 + \cdots) + \underbrace{2(2r_1r_2 + r_1m_2 + r_2m_1) + m_1m_2}_{\mathrm{noise}_{\mathsf{Mult}}}$$

Based on these properties, we can show that

$$\mathsf{Dec}(\bar{c}_1 + \bar{c}_2) = m_1 + m_2 \text{ and } \mathsf{Dec}(\bar{c}_1 \times \bar{c}_2) = m_1 \cdot m_2$$

if the absolute values of $\mathrm{noise}_{\mathsf{Add}}$ and $\mathrm{noise}_{\mathsf{Mult}}$ are less than $p/2$. Note that the noise in the resulting ciphertext increases during homomorphic addition and multiplication (twice and quadratically as much noise as before respectively). If the noise becomes larger than $p/2$, the decryption result of the scheme will be spoiled. As long as the noise is managed, the scheme is able to potentially evaluate all Boolean circuits.

Because managing the noise growth is very expensive and the noise growth induced by multiplication is much larger than that by addition, the performance of homomorphic evaluation is often measured by the maximum *multiplicative depth* of evaluated circuits. The maximum multiplicative depth influences parameters of a HE scheme. Minimizing the multiplicative depth results in not only smaller ciphertexts but also less overall execution time. For example, to support a large number of consecutive multiplications, the secret key p should also be huge in the aforementioned scheme, and it increases overall computational costs. Current FHE schemes are *leveled* (also called *somewhat homomorphic*) in that for fixed encryption parameters they only support computation of a particular depth.[2]

2.2 Boolean Circuit and Multiplicative Depth

Boolean Circuit. A Boolean circuit $c \in \mathbb{C}$ is inductively defined as follows:

$$c \quad \rightarrow \quad \wedge(c, c) \mid \oplus(c, c) \mid x \mid 0 \mid 1$$

where $\wedge$ and $\oplus$ denote AND and XOR respectively, and x denotes an input variable. The grammar is functionally complete because any Boolean functions can be expressed using the grammar. For simplicity, we assume that circuits have a single output value. We will often denote $1 \oplus c$ or $c \oplus 1$ as $\neg c$. In addition, we will use infix notation for $\oplus$ and $\wedge$.

Multiplicative Depth. Let $\ell : \mathbb{C} \rightarrow \mathbb{N}$ be a function that computes the multiplicative depth of a circuit, which is inductively defined as follows:

$$\ell(c) = \begin{cases} 1 + \max_{i \in \{1,2\}} \ell(c_i) & (c = \wedge(c_1, c_2)) \\ \max_{i \in \{1,2\}} \ell(c_i) & (c = \oplus(c_1, c_2)) \\ 0 & (\text{otherwise}) \end{cases}$$

Critical Path. The input-to-output paths with the maximal number of AND gates are called *critical paths*. A set of critical paths, denoted $\mathcal{P}(c)$, of a circuit c is a set of strings over the alphabet of positive integers, which is inductively defined as follows:

[2] A leveled scheme may be turned into a fully homomorphic one by introducing a bootstrapping operation [30], which is computationally heavy.

- If $c = x$ or 1, $\mathcal{P}(c) \stackrel{\text{def}}{=} \{\epsilon\}$, where ϵ is the empty string.
- If $c = f(c_1, c_2)$ where $f \in \{\wedge, \oplus\}$, then

$$\mathcal{P}(c) \stackrel{\text{def}}{=} \bigcup_{i \in \arg\max_{1 \le j \le 2} \ell(c_j)} \{ip \mid p \in \mathcal{P}(c_i)\}$$

A set of critical positions $\mathcal{CP}(c)$ consists of all prefixes of strings in $\mathcal{P}(c)$.

Example 1. *Consider a circuit* $c(v_1, v_2, v_3, v_4)$ *defined as*

$$v_1 \wedge (1 \oplus (v_4 \wedge (1 \oplus (v_2 \wedge v_3)))).$$

The multiplicative depth of c, $\ell(c)$, *is 3 because there are three consecutive AND operations performed on* v_2 *and* v_3. *The set* $\mathcal{P}(c)$ *of critical paths in* c *is*

$$\begin{array}{rcl} \mathcal{P}(c) & = & \{2p \mid p \in \mathcal{P}(1 \oplus (v_4 \wedge (1 \oplus (v_2 \wedge v_3))))\} \\ & = & \{22p \mid p \in \mathcal{P}(v_4 \wedge (1 \oplus (v_2 \wedge v_3)))\} \\ & = & \{222p \mid p \in \mathcal{P}(1 \oplus (v_2 \wedge v_3))\} \\ & = & \{2222p \mid p \in \mathcal{P}(v_2 \wedge v_3)\} \\ & = & \{22221, 22222\} \end{array}$$

The set $\mathcal{CP}(c)$ *of critical positions is:*

$$\{\epsilon, 2, 22, 222, 2222, 22221, 22222\}.$$

Note that in order to decrease the overall multiplicative depth of a Boolean circuit, *all the parallel critical paths of the circuit must be rewritten*. The depth of a critical path can be reduced if we reduce the depth of a sub-circuit at a critical position.

2.3 Problem

Given a Boolean circuit $c \in \mathbb{C}$ whose input variables are $x_1, \cdots, x_n$, we aim to find a semantically equivalent circuit $c' \in \mathbb{C}$ whose depth is smaller than c. Formally, our goal is to find c' such that

$$\forall x_i.\ c(x_1, \cdots, x_n) \iff c'(x_1, \cdots, x_n), \ell(c) > \ell(c'). \quad (1)$$

In this paper, we propose to address this problem by combining program synthesis and term rewriting.

3 Informal Description

In this section, we illustrate our approach with examples. Our approach consists of offline and online phases (Figure 1).

Offline Learning via Program Synthesis. In the offline phase, we use program synthesis to learn a set of rewrite rules from training circuits. Suppose we have the circuit c in Example 1 in the training set:

$$c \stackrel{\text{def}}{=} v_1 \wedge (\neg(v_4 \wedge (\neg(v_2 \wedge v_3)))).$$

The depth of this circuit is 3 and we would like to find a semantically-equivalent circuit c' with a smaller depth (i.e. $\ell(c') \le 2$). To do so, we formulate the task as an instance of the syntax-guided synthesis (SyGuS) problem [2]. The formulation comprises a syntactic specification, in the form of a context-free grammar that constrains the space of possible programs, and a semantic specification, in the form of a logical formula that defines a correctness condition. The syntactic specification for c' is the grammar:

$$\begin{array}{rcl} S & \to & d_2 \\ d_2 & \to & d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\ d_1 & \to & d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\ d_0 & \to & 0 \mid 1 \mid v_1 \mid v_2 \mid v_3 \mid v_4 \end{array}$$

where S denotes the start symbol, and each non-terminal symbol d_i denotes circuits of multiplicative depth $\le i$. The semantic specification for c' is given as a logical formula:

$$\forall v_1, v_2, v_3, v_4.\ c(v_1, v_2, v_3, v_4) \iff c'(v_1, v_2, v_3, v_4)$$

which enforces c' to be semantically equivalent to c. Given this SyGuS formulation, an off-the-shelf program synthesizer (e.g. [3]) is able to find the following circuit c':

$$c' \stackrel{\text{def}}{=} ((\neg(v_3 \wedge v_2)) \wedge (v_1 \wedge v_4)) \oplus v_1$$

which has multiplicative depth 2.

Once we obtain a pair (c, c') of original and optimized circuits, we simplify c and c' by replacing sub-circuits that are equivalent modulo commutativity with a new fresh variable. In this example, $\neg(v_2 \wedge v_3)$ in c and $\neg(v_3 \wedge v_2)$ in c' are equivalent modulo commutativity and therefore we replace them by a new variable x, which simplifies c and c' into $v_1 \wedge (\neg(v_4 \wedge x))$ and $(x \wedge (v_1 \wedge v_4)) \oplus v_1$, respectively. Note that the simplified circuits are still semantically equivalent. We replace sub-circuits with a variable after we check for equivalence using a SAT solver.

The purpose of this simplification step is to generalize the knowledge and maximize the possibility of applying the rewrite rule for optimization in the online phase. However, care is needed not to over-generalize and destroy the syntactic structures of the circuits. For example, if we aim to replace all semantically equivalent sub-circuits with a new fresh variable, we would obtain $x \iff x$, which is useless.

In summary, the offline learning phase produces the following rewrite rule:

$$v_1 \wedge (\neg(v_4 \wedge x)) \to (x \wedge (v_1 \wedge v_4)) \oplus v_1. \quad (2)$$

Online Optimization via Term Rewriting. In the online phase, we use the learned rewrite rule to optimize unseen circuits. Suppose we want to optimize the following circuit whose multiplicative depth is 4:

$$((v_5 \wedge v_6) \wedge (\neg((v_7 \wedge v_8) \wedge (\neg((v_8 \wedge v_9) \wedge (v_9 \wedge v_7)))))). \quad (3)$$

To optimize the circuit, we first compare it with the left-hand side of the learned rewrite rule (i.e. $v_1 \wedge (\neg(v_4 \wedge x))$), and find a substitution σ that makes the two circuits equivalent. For example, our matching algorithm in Section 4 is able to find the following substitution:

$$\sigma = \left\{ \begin{array}{rcl} v_1 & \mapsto & v_5 \wedge v_6 \\ v_4 & \mapsto & v_7 \wedge v_8 \\ x & \mapsto & (\neg(v_8 \wedge v_9) \wedge (v_9 \wedge v_7)) \end{array} \right\}.$$

Note that $\sigma(v_1 \wedge (\neg(v_4 \wedge x)))$ is equivalent to the circuit in (3). Next, we apply the substitution to the right-hand side of the rewrite rule, obtaining the following optimized circuit:

$$(\neg((v_8 \wedge v_9) \wedge (v_9 \wedge v_7)) \wedge ((v_5 \wedge v_6) \wedge (v_7 \wedge v_8))) \oplus (v_5 \wedge v_6).$$

whose multiplicative depth is 3. In our approach, the resulting circuit is guaranteed to be semantically equivalent to the original one in (3).

Scaling via Divide-and-Conquer. We have described how to obtain a rewrite rule from a small circuit and applying it into a new yet small circuit. In practice, however, real circuits are much larger, and the aforementioned method using the SyGuS formulation is not directly applicable. Even state-of-the-art SyGuS tools can only handle small circuits because the search space for synthesis grows exponentially with the maximum depth and number of input variables.

To address this scalability issue, we apply our approach in a divide-and-conquer manner; we divide a circuit into pieces, find a replacement for each piece, and finally compose them to form a final circuit. For example, consider the circuit c_{ex} of depth 5, which is depicted in Figure 2(a) (the critical path is highlighted in red):

$$c_{ex} \stackrel{\text{def}}{=} ((((a \wedge b) \wedge c) \wedge d) \wedge e) \wedge f. \tag{4}$$

We can divide the circuit into two pieces r_1 and r_2 through which a critical path passes. By introducing two auxiliary variables, c_{ex} can be rewritten as r_2 where

$$r_2 \stackrel{\text{def}}{=} (r_1 \wedge e) \wedge f, \quad r_1 \stackrel{\text{def}}{=} ((a \wedge b) \wedge c) \wedge d.$$

We separately reduce the depths of r_1 and r_2 in order. We first find a replacement for r_1. We can replace r_1 of depth 3 by the following:

$$r_1' \stackrel{\text{def}}{=} (a \wedge b) \wedge (c \wedge d)$$

which has depth 2 and the same semantics as r_1. Next, we find a replacement for r_2. We treat r_1 in the definition of r_2 as a special variable that has its own depth 2. Considering the depth of r_1, we replace r_2 of depth 4 by

$$r_2' \stackrel{\text{def}}{=} r_1' \wedge (e \wedge f)$$

that has depth 3 and the same semantics as r_2. Combining r_1' and r_2' produces the final circuit of depth 3. We use this divide-and-conquer strategy in both of our offline learning and online rewriting phases.

4 Algorithm

We first review (Section 4.1) key definitions and results borrowed from [6] that will be used in the rest of the paper. Then we present the offline learning phase (Section 4.2) and online optimization phase (Section 4.3) based on term rewriting.

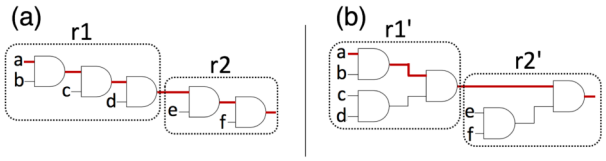

Figure 2. (a) The circuit c_{ex} of depth 5. (b) A circuit that has depth 3 and the same semantics as c_{ex}.

4.1 Preliminaries

Term. A signature Σ is a set of function symbols, where each $f \in \Sigma$ is associated with a non-negative integer n, the arity of f (denoted $arity(f)$). For $n \geq 0$, we denote the set of all n-ary elements Σ by $\Sigma^{(n)}$. Function symbols of 0-arity are called constants. Let X be a set of variables. The set $T_{\Sigma,X}$ of all Σ-terms over X is inductively defined; $X \subseteq T_{\Sigma,X}$ and $\forall n \geq 0, f \in \Sigma^{(n)}.\ t_1, \cdots, t_n \in T_{\Sigma,X}.\ f(t_1, \cdots, t_n) \in T_{\Sigma,X}$. We will denote $Var(s)$ for $s \in T_{\Sigma,X}$ as a set of variables in term s. Note the set $\mathbb{C}$ of circuits consists of terms over $\Sigma = \{\wedge, \oplus, 0, 1\}$.

Position. The set of positions of term s is a set $Pos(s)$ of strings over the alphabet of positive integers, which is inductively defined as follows:

- If $s = x \in X$, $Pos(s) \stackrel{\text{def}}{=} \{\epsilon\}$.
- If $s = f(s_1, \cdots, s_n)$, then $Pos(s) \stackrel{\text{def}}{=} \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(s_i)\}$.

The position ϵ is called the root position of term s. The size $|s|$ of term s is the cardinality of $Pos(s)$. For $p \in Pos(s)$, the subterm of s at position p, denoted by $s\ |_p$, is defined by induction on the length of p: (i) $s\ |_\epsilon \stackrel{\text{def}}{=} s$ and (ii) $f(s_1, \cdots, s_n)\ |_{iq} \stackrel{\text{def}}{=} s_i\ |_q$. For $p \in Pos(s)$, we denote by $s[p \leftarrow t]$ the term that is obtained from s by replacing the subterm at position p by t. Formally,

- $s[\epsilon \leftarrow t] \stackrel{\text{def}}{=} t$
- $f(s_1, \cdots, s_n)[iq \leftarrow t] \stackrel{\text{def}}{=} f(s_1, \cdots, s_i[q \leftarrow s], \cdots, s_n)$.

Substitution. A $T_{\Sigma,X}$-substitution is a function $X \to T_{\Sigma,X}$. The set of all $T_{\Sigma,X}$-substitutions is denoted by $Sub(T_{\Sigma,X})$. Any $T_{\Sigma,X}$-substitution σ can be extended to a mapping $\hat{\sigma} : T_{\Sigma,X} \to T_{\Sigma,X}$ as follows: for $x \in X$, $\hat{\sigma}(x) \stackrel{\text{def}}{=} \sigma(x)$ and for any non-variable term $s = f(s_1, \cdots, s_n)$, $\hat{\sigma}(s) \stackrel{\text{def}}{=} f(\hat{\sigma}(s_1), \cdots, \hat{\sigma}(s_n))$. With a slight of abuse of notation, we denote $\hat{\sigma}$ as just σ.

Term Rewriting. A Σ-identity (or simply identity) is a pair $\langle s, t \rangle \in T_{\Sigma,X} \times T_{\Sigma,X}$. Identities will be written as $s \approx t$. A term rewrite rule is an identity $\langle l, r \rangle$, written $l \to r$, such that $l \notin X$ and $Var(r) \subseteq Var(l)$. A term rewriting system $\langle \Sigma, E \rangle$ consists of a set Σ of function symbols and a set E of term rewrite rules over $T_{\Sigma,X}$. We will often identify such a system with its rule set E, leaving Σ implicit. The rewrite relation $\to_E$ on $T_{\Sigma,X}$ induced by a term rewriting system E

is defined as follows:

$$s \to_E t \iff \exists l \to r \in E, p \in Pos(s), \sigma \in Sub(T_{\Sigma,X}). \; s|_p = \sigma(l), t = s[p \leftarrow \sigma(r)]$$

Example 2. *Let* $E \stackrel{\text{def}}{=} \{x \wedge (y \wedge z) \approx (x \wedge y) \wedge z, 1 \wedge x \approx x, x \wedge y \approx y \wedge x\}$. *Then,* $1 \wedge (a \wedge 1) \to_E (1 \wedge a) \wedge 1 \to_E a \wedge 1 \to_E 1 \wedge a \to_E a$.

Equational Theory. Let $\leftrightarrow_E^*$ denote the reflexive-transitive-symmetric closure of $\to_E$. The identity $s \approx t$ is a semantic consequence of E (denoted $E \models s \approx t$) iff $s \leftrightarrow_E^* t$. And the relation $\approx_E \stackrel{\text{def}}{=} \{\langle s, t\rangle \in T_{\Sigma,X} \times T_{\Sigma,X} \mid E \models s \approx t\}$ is called the *equational theory* induced by E.

Example 3. *The theory of commutativity for circuits is* $\approx_C \stackrel{\text{def}}{=} \{\langle s, t\rangle \in \mathbb{C}\times\mathbb{C} \mid C \models s \approx t\}$ *where* $C = \{x \wedge y \approx y \wedge x, x \oplus y \approx y \oplus x\}$.

Example 4. *Boolean ring theory is* $\approx_{\mathcal{R}} \stackrel{\text{def}}{=} \{\langle s, t\rangle \in \mathbb{C} \times \mathbb{C} \mid \mathcal{R} \models s \approx t\}$ *where*

$$\mathcal{R} = \left\{ \begin{array}{cc} x \oplus y \approx y \oplus x, & x \wedge y \approx y \wedge x, \\ (x \oplus y) \oplus z \approx x \oplus (y \oplus z), & \\ (x \wedge y) \wedge z \approx x \wedge (y \wedge z), & \\ x \oplus x \approx 0, & x \wedge x \approx x, \\ 0 \oplus x \approx x, & 0 \wedge x \approx 0, \\ x \wedge (y \oplus z) \approx (x \wedge y) \oplus (x \wedge z), & \\ 1 \wedge x \approx x & \end{array} \right\}$$

Boolean ring theory formalizes digital circuits. For any two circuits c_1, c_2, $c_1 \leftrightarrow_{\mathcal{R}}^* c_2$ means they are semantically equivalent due to Birkhoff [6].

Example 5. *The original circuit* c *and its optimized version* c' *in Section 3 are semantically equivalent because*

$$\begin{array}{l} c = v_1 \wedge (1 \oplus (v_4 \wedge (\neg(v_2 \wedge v_3)))) \\ \to_{\mathcal{R}} v_1 \oplus (v_1 \wedge (v_4 \wedge (\neg(v_2 \wedge v_3)))) \\ \to_{\mathcal{R}} v_1 \oplus ((v_1 \wedge v_4) \wedge (\neg(v_3 \wedge v_2))) \\ \to_{\mathcal{R}} ((v_1 \wedge v_4) \wedge (\neg(v_3 \wedge v_2))) \oplus v_1 \\ \to_{\mathcal{R}} ((\neg(v_3 \wedge v_2)) \wedge (v_1 \wedge v_4)) \oplus v_1 = c' \end{array}$$

***E*-Matching.** A substitution σ is a E-matcher of two terms s and t if $\sigma(s) \approx_E t$. Given two terms s and t, a E-matching algorithm computes $\{\sigma \in Sub(T_{\Sigma,X}) \mid \sigma(s) \approx_E t\}$.

Example 6. *Given two terms* $s = x \wedge y$ *and* $t = (a \wedge b) \wedge (b \wedge a)$, *C-matching algorithm returns two substitutions which are* $\{x \mapsto a \wedge b, y \mapsto b \wedge a\}$ *and* $\{x \mapsto b \wedge a, y \mapsto a \wedge b\}$.

4.2 Learning Rewrite Rules

In this section, we describe how to learn rewrite rules using the divide-and-conquer approach described in Section 3. The method is inspired by the prior work [26], which uses syntax-guided synthesis to automatically transform a circuit into an equivalent and provably secure one.

Algorithm 1 Synthesis-based Rule Learning

Input: c: input boolean circuit
Input: θ: threshold for termination condition
Input: n: predefined size limit for chosen regions
Output: E: a set of rewrite rules

```
1:  c' ← c
2:  E ← ∅
3:  w ← CP(c')
4:  while w ≠ ∅ and |c'|/|c| < θ do
5:      remove a pos from w
6:      ⟨r, σ⟩ ← GetRegion(c' |_pos, n)
7:      r' ← Synthesize(r, ℓ(r) − 1, σ)
8:      if r' ≠ ⊥ then
9:          E ← E ∪ {Normalize(r → r')}
10:         c' ← c'[pos ← σ(r')]
11:         w ← CP(c')
12:     end if
13: end while
14: return E
```

4.2.1 The Overall Algorithm.

The pseudocode is shown in Algo. 1. Here, c denotes an original training circuit, θ denotes a threshold value for termination condition, n is an user-provided predefined limit for region selection. The algorithm generates an optimized circuit c', and returns a set E of rewrite rules collected in the process of optimization.

Our algorithm repeatedly identifies a circuit region and synthesizes a replacement. To identify a circuit region, we randomly choose a critical path and traverse the path from input-to-output. If the left and right children at a position have different depths, we include both gates in fan-in and recurse on the child of deeper depth. We repeat this process until the region size reaches a predefined limit. Once we successfully synthesize a replacement, we can decrease the overall depth if a unique critical path passes through the region. Otherwise, we decrease the number of parallel critical paths by one.

Our method first initializes c' to be the original circuit c, E to be the empty set and the worklist w to be a set of critical positions, respectively (lines 1–3). The loop (lines 4 – 13) repeats the process of selecting a region and synthesizing a replacement. First, a critical position *pos* is chosen in the input-to-output order (line 5). Given a subcircuit at *pos*, the GetRegion procedure is invoked to obtain a circuit region r such that $|r| \le n$ (line 6). The GetRegion procedure substitutes some subterms of a given circuit with fresh variables and returns the result along with the substitution. Section 4.2.2 will detail more on this procedure. Next, we invoke a SyGuS solver to synthesize a replacement for r (line 7). If a solution is found (line 8), we obtain a term rewrite rule $r \to r'$. We generalize the rule by invoking the Normalize procedure, and add it into the set E (line 9). The old region r is replaced with the new region r' (line 10). Because the replacement step may change the overall structure of the

Algorithm 2 GetRegion

Input: c: input boolean circuit region
Input: n: predefined size limit for regions
Output: r: a circuit region
Output: σ: a substitution from variables to circuits
1: $x \leftarrow$a new fresh variable
2: **if** $n = 1$ or $|c| = 1$ **then**
3: **return** $\langle x, \{x \mapsto c\}\rangle$
4: **end if**
5: $c_1 \leftarrow c\,|_1$
6: $c_2 \leftarrow c\,|_2$
7: **if** $\ell(c_1) > \ell(c_2)$ **then**
8: $\langle r', \sigma\rangle \leftarrow$ GetRegion$(c_1, n-1)$
9: **return** $\langle c[1 \leftarrow r', 2 \leftarrow x], \sigma\{x \mapsto c_2\}\rangle$
10: **else**
11: $\langle r', \sigma\rangle \leftarrow$ GetRegion$(c_2, n-1)$
12: **return** $\langle c[1 \leftarrow x, 2 \leftarrow r'], \sigma\{x \mapsto c_1\}\rangle$
13: **end if**

current circuit, we recompute critical positions and update the worklist (line 11). This process is repeated as long as there is room for improvement, and the ratio between the sizes of c' and c does not exceed the threshold value θ (line 4). The ratio between the circuit sizes is considered because the depth reduction may not be beneficial if a new circuit c' additionally performs a huge number of AND/XOR operations. Although the multiplicative depth is the dominating factor for homomorphic evaluation performance, the number of operations can also have a non-trivial impact if it is enormous. The threshold value varies depending on the underlying FHE schemes. In our evaluation, we set θ to be 3. The algorithm eventually returns the set E of rewrite rules (line 14), which include all the transformations occurred while optimizing c into c'.

4.2.2 Region Selection.

The GetRegion procedure for the region selection is shown in Algo. 2. The region selection method is a heuristic based on our observation that replacing long and narrow regions covering critical paths often leads to significant optimization effects. If the given region size n is 1 or the given circuit region is a variable of a constant (i.e., $|c| = 1$) (line 2), we just represent the given circuit region as a fresh variable and return it along with the corresponding substitution (line 3). Otherwise (i.e., $|c| > 1$), we first let c_1 and c_2 be the left and right child of c, resp. (lines 5 – 6). If the depth of c_1 (c_2, resp.) is deeper than the other (line 7 (line 10, resp.)), we keep extending the region in c_1 (c_2, resp.) (line 8 (line 11, resp.)), and substitute c_2 (c_1, resp.) with a fresh variable (line 9 (line 12, resp.)).

Example 7. *Consider the circuit c_{ex} in (4).* GetRegion$(c_{ex}, 5)$ *returns* $\langle (r_1 \wedge e) \wedge f, \{r_1 \mapsto ((a \wedge b) \wedge c) \wedge d\}\rangle$ *(see Fig. 2(a)).*

4.2.3 Synthesizing Replacement.

Given a circuit region r, an upper bound n of desired multiplicative depths, and a substitution σ, the function Synthesize returns a new semantically equivalent region r' of depth $\leq n$.

For $1 \leq i \leq n$, let x^i denote one of variables such that $\ell(\sigma(x^i)) = i$. We can formulate a SyGuS instance as follows. The syntactic specification for r' is

$$\begin{array}{rcl} S & \rightarrow & d_n \\ d_n & \rightarrow & d_{n-1} \wedge d_{n-1} \mid d_n \oplus d_n \mid d_{n-1} \mid x^n \\ d_{n-1} & \rightarrow & d_{n-2} \wedge d_{n-2} \mid d_{n-1} \oplus d_{n-1} \mid d_{n-2} \mid x^{n-1} \\ & \vdots & \\ d_0 & \rightarrow & 0 \mid 1 \mid x^0 \end{array}$$

where S denotes the start symbol and each d_i represents circuits of multiplicative depth $\leq i$. The semantics specification for r' enforces the equivalence of r and r':

$$\forall x^0, \cdots, x^{n-1}.\ r(x^0, \cdots, x^{n-1}) \iff r'(x^0, \cdots, x^{n-1}).$$

When Synthesize fails to find a solution, it returns $\perp$.

Example 8. *After selecting the region r_2 as in Example 7, we find a replacement for r_2 using the following formulation, hoping to reduce the depth from 5 to 4. The syntactic specification for r_2' is*

$$\begin{array}{rcl} S & \rightarrow & d_4 \\ d_4 & \rightarrow & d_3 \wedge d_3 \mid d_4 \oplus d_4 \mid d_3 \\ d_3 & \rightarrow & d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \mid r_1 \\ d_2 & \rightarrow & d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\ d_1 & \rightarrow & d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\ d_0 & \rightarrow & 0 \mid 1 \mid e \mid f \end{array}$$

and the semantics specification is the semantic equivalence with r_2. Note that r_1 is producible from d_3 because its depth is 3. Given this problem, a SyGuS solver (e.g., [3]) finds the solution $r_1 \wedge (e \wedge f)$ which has depth 4.

4.2.4 Collecting and Simplifying Rewrite Rules.

When we obtain a rewrite rule $l \rightarrow r$, we simplify it by invoking the Normalize procedure (line 9 in Algo. 1). We normalize each rewrite rule $l \rightarrow r \in E$ as follows:

- Let $\mathcal{S} = \{(l|_{p_l}, r|_{p_r}) \mid p_l \in Pos(l), p_r \in Pos(r), l|_{p_l} \approx_C r|_{p_r}\}$.
- For each $(l|_{p_l}, r|_{p_r}) \in \mathcal{S}$, we transform $l \rightarrow r$ into $l' \rightarrow r'$ where $l' = \sigma(l), r' = \sigma(r), \sigma = \{l|_{p_l} \mapsto x, r|_{p_r} \mapsto x\}$, and x is a fresh variable. We transform the rule only if l' is semantically equivalent to r'.

We consider a term rewrite rule that cannot be further simplified by this procedure *normalized modulo commutativity.*

Example 9. *Suppose we want to normalize a rewrite rule:*

$$\underbrace{(a \wedge b) \wedge a}_{l} \rightarrow \underbrace{(b \wedge a) \wedge b}_{r}.$$

Note that $l\ |_1 = (a \wedge b) \approx_C r\ |_1 = (b \wedge a)$. If we replace the subterms $l\ |_1$ and $r\ |_1$ with a fresh variable x, we would obtain $x \wedge a \rightarrow x \wedge b$, which is undesirably semantics-changing. In this case, we do not replace the subterms.

4.3 Rule-based Circuit Optimization

Next, we describe our algorithm that uses the set E of (normalized) learned rewrite rules to optimize unseen circuits.

4.3.1 Our Term Rewriting System. Our term rewriting system is based on the following relation $\rightarrow_{E,\ell}$ induced by E (learned rewrite rules) and ℓ (the function computing the multiplicative depth).

$$s \rightarrow_{E,\ell} t \iff \exists l \rightarrow r \in E, p \in \mathcal{CP}(s), \sigma \in Sub(\mathbb{C}). \; s \mid_p \approx_C \sigma(l), \ell(\sigma(l)) > \ell(\sigma(r)), t = s[p \leftarrow \sigma(r)].$$

Because our primary goal is to reduce the overall multiplicative depth, the above rewrite relation differs from the ordinary relation in Section 4.1 in three aspects.

First, we rewrite critical paths by considering only critical positions $\mathcal{CP}(s)$ of a given circuit s. Rewriting non-critical paths are not of our interest.

Second, we admit a rewrite step only if it decreases the depth of a critical path. This condition is reflected in $\ell(\sigma(l)) > \ell(\sigma(r))$.

Lastly, we perform rewriting modulo commutativity to provide flexibility to the rewriting procedure. This is for maximizing the possibility of applying the learned rewrite rules for optimization. Instead of syntactically matching a left-hand side of a rule with a subterm as in the ordinary rewrite relation, each rewrite step requires C-matching, which is reflected in $s \mid_p \approx_C \sigma(l)$. Here, a complication arises that there may be multiple C-matchers. In such a case, we choose the one that can reduce depth.

Example 10. *Recall the rewrite rule (2) in Section 3*

$$\underbrace{v_1 \wedge (\neg(v_4 \wedge x))}_{l} \rightarrow \underbrace{(x \wedge (v_1 \wedge v_4)) \oplus v_1}_{r}.$$

and the target circuit (3) of depth 4

$$(v_5 \wedge v_6) \wedge (\neg((v_7 \wedge v_8) \wedge (\neg((v_8 \wedge v_9) \wedge (v_9 \wedge v_7))))).$$

There are two substitutions that make l match with the target circuit: $\sigma_1 = \{v_1 \mapsto v_5 \wedge v_6, v_4 \mapsto v_7 \wedge v_8, x \mapsto (\neg(v_8 \wedge v_9) \wedge (v_9 \wedge v_7))\}$ *and* $\sigma_2 = \{v_1 \mapsto v_5 \wedge v_6, v_4 \mapsto (\neg(v_8 \wedge v_9) \wedge (v_9 \wedge v_7)), x \mapsto v_7 \wedge v_8\}$. *Applying the substitutions into r gives us two candidates for the replacement, which are*

$$\sigma_1(r) = (\neg((v_8 \wedge v_9) \wedge (v_9 \wedge v_7)) \wedge ((v_5 \wedge v_6) \wedge (v_7 \wedge v_8))) \oplus (v_5 \wedge v_6),$$
$$\sigma_2(r) = ((v_7 \wedge v_8) \wedge ((v_5 \wedge v_6) \wedge (\neg(v_8 \wedge v_9) \wedge (v_9 \wedge v_7)))) \oplus (v_5 \wedge v_6).$$

Note that $\sigma_1(r)$ has depth 3 whereas $\sigma_2(r)$ has depth 4. Because only σ_1 can reduce the depth, we choose σ_1.

The following theorems ensure that our term rewriting system is semantics-preserving and terminating.

Theorem 1 (Soundness). $\forall c, c' \in \mathbb{C}.\ c \rightarrow_{E,\ell} c' \Rightarrow c \approx_{\mathcal{R}} c'$.

Proof. Available in supplementary material. □

Theorem 2 (Termination). $\rightarrow_{E,\ell}$ *is a terminating relation.*

Proof. Available in supplementary material. □

Intuitively, termination is enforced because every rewrite step decreases the depth of a critical path. If the rewritten critical path is unique, we reduce the overall multiplicative depth of the circuit. Otherwise, we reduce the number of parallel critical paths. Because every circuit has at least one critical path of non-negative depth, the rewriting procedure eventually terminates.

Using the rewrite relation $\rightarrow_{E,\ell}$, given a circuit c, we perform term rewriting on c to obtain an optimized circuit c' such that $c \overset{*}{\rightarrow}_{E,\ell} c'$. At each rewrite step, we randomly choose a critical path and traverse the path to find a target region to be replaced. The traversal order is randomly chosen between the input-to-output and output-to-input orders. Similarly to Algo. 1, we stop the rewriting procedure when there are so many additional AND/XOR gates in c' that the depth reduction may not be beneficial.

4.3.2 Optimizations. In practice, we apply the following optimization techniques into the rewriting procedure.

Prioritizing Large Rewrite Rules. In the case where multiple rewrite rules are applicable, we choose the largest rule. The size of a rule $l \rightarrow r$ is simply measured by $|l|$. This heuristic is based on our observation that large rules are applicable less often than small rules, but they expedite transformation by modifying a wider area.

Bounded C-matching. From a performance perspective, the main weakness of our rewriting system is that each rewrite step requires C-matching, which is known to be NP-complete [36]. We limit the search space of C-matching algorithm by limiting the number of applications of commutativity rules (see supplementary material for details).

Term Graph Rewriting. So far, we have presented our method as if circuits are represented as functional expressions for ease of presentation. In practice, we cannot directly implement this kind of conventional term rewriting based on strings or trees because of an efficiency issue. For example, term rewrite rules such as (2) containing some variable more often on its right-hand side than on its left-hand side can increase the size of a term by a non-constant amount. This problem can be overcome by creating several pointers to a subterm instead of copying it.

For efficiency, we conduct *term graph rewriting* [41] on circuits. Term graph rewriting is a model for computing with graphs representing functional expressions. Graphs allow sharing common subterms, which improves the efficiency of conventional term rewriting in space and time. Thus, we represent circuits as graphs and perform rewriting on the graphs by translating term rewrite rules into suitable graph transformation rules. Term graph rewriting is sound with respect to term rewriting in that every graph transformation step corresponds to a sequence of applications of term rewrite rules. The interested reader is referred to [41] for

more details about the soundness proof and the translation method.

5 Evaluation

We implemented our method as a tool named Lobster[3]. This section evaluates our Lobster system to answer the questions:

- **Q1:** How effective is Lobster for optimizing FHE applications from various domains?
- **Q2:** How does Lobster compare with existing general-purpose FHE optimization techniques?
- **Q3:** What is the benefit of the rule normalization and equational matching for rule applications of Lobster?
- **Q4:** What is the benefit of using learned rewrite rules?
- **Q5:** How sensitive is Lobster to changes in a given training set?

All of our experiments were conducted on Linux machines with Intel Xeon 2.6GHz CPUs and 256G of memory.

5.1 Experimental Setup

Implementation. Lobster comprises three pieces: (i) an offline rule learner, (ii) an online circuit optimizer, and (iii) a homomorphic circuit evaluator. Lobster is written in OCaml and consists of about 3K lines of code.

The offline rule learner uses EUSolver [3], which is an open-source search-based synthesizer. We chose EUSolver among the general-purpose synthesizers that participated in the 2019 SyGuS competition [46] since the tool performs best for our optimization tasks. We use a timeout of one hour for synthesizing each rewrite rule.

The online circuit optimizer transforms Boolean circuits generated by Cingulata [15], an open-source FHE compiler, into depth-optimized ones. Cingulata first directly translates a given FHE application written in C++ into a Boolean circuit representation, and then heuristically minimizes the circuit area by removing redundancy using the ABC tool [12], which has been widely used for hardware synthesis. Then, our optimizer performs the rewriting procedure on the resulting circuit.

Circuits optimized by the online optimizer are evaluated by our homomorphic circuit evaluator built using HElib [32].[4] When homomorphically evaluating circuits, we set the security parameter to 128 which is usually considered large enough. It means a ciphertext can be broken in the worst case time proportional to 2^{128}.

Benchmarks. First, our benchmarks comprise 25 FHE applications written using the Cingulata APIs shown in Table 1. We first collected 64 benchmarks from the following three sources and ruled out 21 which are already depth-optimal and 18 which are out of reach for homomorphic evaluation because of the enormous circuit sizes.

- Cingulata benchmarks – 9 FHE-friendly algorithms from diverse domains (medical diagnosis, stream cipher, search, sort) available in [21].
- Sorting benchmarks – four privacy-preserving sorting algorithms (merge, insertion, bubble, and odd-even) presented in [17].
- Hackers Delight benchmarks – 26 homomorphic bitwise operations adapted from [52][5], a collection of bit-twiddling hacks. We include these benchmarks because they can be potentially used as building blocks for efficient FHE applications that perform computations over fixed-width integers.
- EPFL benchmarks – 25 circuits from EPFL combinational benchmark suite [1]. The circuits are intendedly suboptimal to test the ability of circuit optimization tools.

We concluded the 21 benchmarks are depth-optimal based on empirical evidence. We could not mine any rules from their circuit representations even after 7 days of running the offline learner. This means that even the state-of-the-art synthesizer with practically unlimited time cannot find any improvement. 7 out of 9 Cingulata benchmarks fall into this category since they are already carefully hand-tuned to be depth-optimal. 14 out of 26 Hackers Delight benchmarks have no room for improvement because their circuit representations are fairly simple. We excluded 18 circuits from EPFL benchmarks because they are out of reach for homomorphic evaluation even with the state-of-the-art FHE scheme [32]. Our homomorphic circuit evaluator runs out of memory for the circuits where the number of AND/XORs is greater than 10,000, or the multiplicative depth is larger than 100. All the FHE sorting algorithms can take up to 6 encrypted 8-bit integers as input.

Baseline. We compare Lobster to the work by Carpov et al. [14], which also aims at minimizing the multiplicative depth of circuits for homomorphic evaluation. The work is also based on term rewriting, but only with two hand-written rewrite rules. The first rule is based on AND associativity: $(x \wedge y) \wedge z \rightarrow x \wedge (y \wedge z)$. In a given circuit c, a substitution σ such that $\sigma((x \wedge y) \wedge z)$ is syntactically matched with a subcircuit of c is found. The matched part $\sigma((x \wedge y) \wedge z)$ is replaced with $\sigma(x \wedge (y \wedge z))$ if $\ell(y) < \ell(x)$ and $\ell(z) < \ell(x)$. This rewrite rule, when applied into a critical path, reduces the depth by one from $\ell(\sigma(x)) + 2$ to $\ell(\sigma(x)) + 1$. The second rewrite rule is based on XOR distributivity: $(x \oplus y) \wedge z \rightarrow (x \wedge z) \oplus (y \wedge z)$. This rule does not affect the depth, but it can make the first rule applicable by clearing XOR operators away. The two rewrite rules repeatedly rewrite critical paths until a heuristic

[3] **L**earning to **O**ptimize **B**oolean circuits using **S**ynthesis & **TE**rm **R**ewriting

[4] We could not use the homomorphic circuit evaluator provided by Cingulata because it crashed for some of our evaluation benchmarks, which are fairly sizeable circuits.

[5] 22 benchmarks used for program synthesis [35] + 4 excerpted from [52]

Table 1. Benchmark characteristics. ×**Depth** denotes the multiplicative depth. **#AND** and **Size** give the number of AND operations and the circuit size, respectively.

Name	Description	×Depth	#AND	Size
cardio	medical diagnostic algorithm [16]	10	109	318
dsort	FHE-friendly direct sort [17]	9	708	1464
msort	merge sort [17]	45	810	1525
isort	insertion sort [17]	45	810	1525
bsort	bubble sort [17]	45	810	1525
osort	oddeven sort [17]	25	702	1343
hd-01	isolate the rightmost 1-bit [35]	6	87	118
hd-02	absolute value [35]	6	76	229
hd-03	floor of average of two integers (a clever impl.) [35]	5	27	64
hd-04	floor of average of two integers (a naive impl.) [52]	10	75	159
hd-05	max of two integers [35]	7	121	295
hd-06	min of two integers [35]	7	121	295
hd-07	turn off the rightmost contiguous string of 1-bits [35]	5	17	32
hd-08	determine if an integer is a power of 2 [35]	6	18	37
hd-09	round up to the next highest power of 2 [35]	14	134	236
hd-10	find first 0-byte [52]	6	35	73
hd-11	the longest length of contiguous string of 1-bits [52]	18	391	652
hd-12	number of leading 0-bits [52]	16	116	232
bar	barrel shifter [1]	12	3141	5710
cavlc	coding-cavlc [1]	16	655	1219
ctrl	ALU control unit [1]	8	107	180
dec	decoder [1]	3	304	312
i2c	i2c controller [1]	15	1157	1987
int2float	int to float converter [1]	15	213	386
router	lookahead XY router [1]	19	170	277

termination condition is satisfied. As the tool is not publicly available, we reimplemented their algorithm.[6]

5.2 Effectiveness of Lobster

Optimization Effect. We evaluate Lobster on the benchmarks and compare it with Carpov et al. [14]. Both of the tools are provided circuits initially generated by Cingulata. We aim to determine whether Lobster can learn rewrite rules from training circuits and effectively generalize them for optimizing other unseen circuits. To this end, we conduct ***leave-one-out cross validation***; for each benchmark, we use rewrite rules learned from the other remaining benchmarks. Both of the tools are given the timeout limit of 12 hours for the optimization tasks; in case of exceeding the limit, we use the best intermediate results computed so far. We measure reduction ratios of the multiplicative depth and speedups in overall homomorphic evaluation time (vs. initial Cingulata-generated circuits).

The results are summarized in Fig. 3. More detailed information can be found in Table 2. Lobster is able to optimize 19 out of 25 benchmarks within the timeout limit. Lobster achieves 1.18x – 3.71x speedups with the geometric mean of 2.05x. The number of AND gates increases up to 2x more with the geometric mean of 1.17x. The depth reduction ratios range from 6.3% to 47.4% with the geometric mean of 21.9%. Note that the depth reduction ratios are generally proportional to performance improvements (but not exactly proportional since the number of AND operations also influences the performance).

We next study the results in detail. Most notably, Lobster achieves 2.45x and 1.34x speedups for the two Cingulata benchmarks cardio and dsort, respectively. Recall that they are already carefully hand-tuned to be depth optimized. This result shows that our method provides significant performance gains that are complementary to those achieved by domain-specific optimizations. The four sorting benchmarks also observe significant performance improvements. Lobster reduces the depth by 20% for each of them. The osort benchmark shows a 2.4x speedup, and the other three benchmarks show 1.5x speedups. As of the Hacker's Delight benchmarks, 7 out of 12 observe improvements. For hd-09, hd-11 and hd-12, we observe 1.32x – 1.64x speedups. In particular, the

[6] We use the "random" priority function because it slightly outperforms the "non-random" heuristics according to the results in [14].

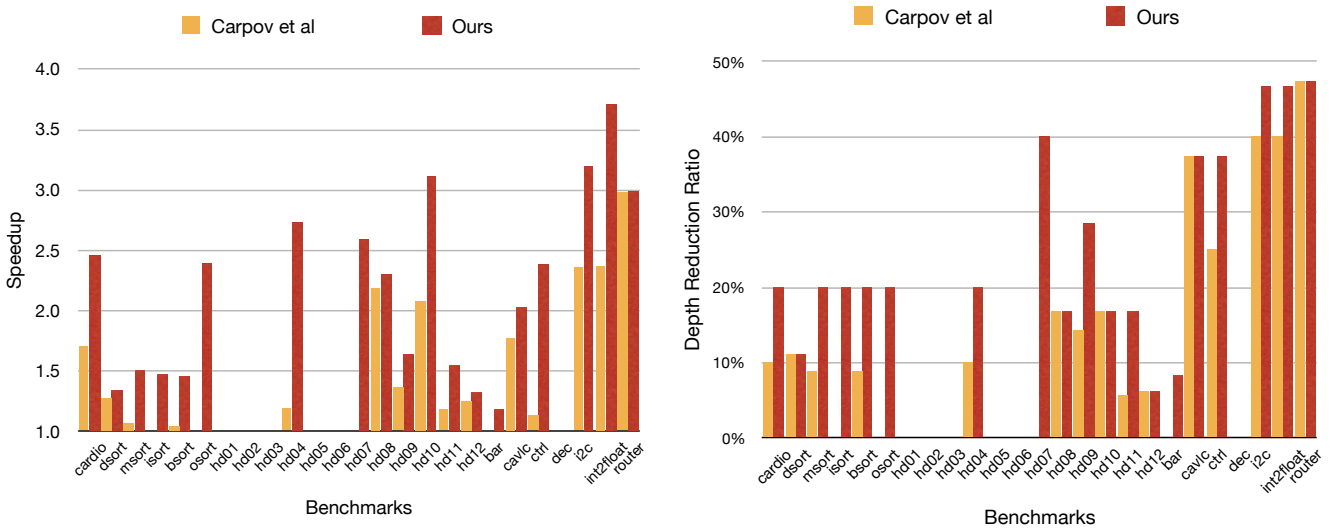

Figure 3. Main results comparing the optimization performance of Lobster and Carpov et al [14] – Speedups in overall homomorphic evaluation time (left) (vs. initial Cingulata-generated circuits) and depth reduction ratios (right).

Table 2. Detailed main results. **Opt. Time** gives online optimization time. The timeout for optimization is set to 12 hours. **#AND ↑** shows the ratio between the number of AND gates of the optimized circuit and the original one. **Eval. Time** shows homomorphic evaluation time (where '-' means that the evaluation time is the same as the original).

	Original		Carpov et al [14]				Lobster			
Name	**×Depth**	**Eval. Time**	**×Depth**	**Opt. Time**	**#AND ↑**	**Eval. Time**	**×Depth**	**Opt. Time**	**#AND ↑**	**Eval. Time**
cardio	10	17m 14s	9	10s	x1.07	10m 08s	8	24s	x1.06	7m 02s
dsort	9	10m 52s	8	1m 25s	x1.08	8m 29s	8	2m 31s	x1.12	8m 08s
msort	45	5h 20m 59s	41	19m 39s	x1.02	5h 00m 06s	36	8h 02m 06s	x1.79	3h 33m 19s
isort	45	5h 20m 16s	45	5m 06s	-	-	36	8h 29m 58s	x1.83	3h 38m 13s
bsort	45	5h 21m 46s	41	19m 04s	x1.02	5h 06m 09s	36	7h 59m 44s	x1.83	3h 39m 57s
osort	25	2h 16m 58s	25	1m 55s	-	-	20	3h 22m 20s	x2.00	57m 06s
hd-01	6	4m 36s	6	1s	-	-	6	13s	-	-
hd-02	6	4m 50s	6	1s	-	-	6	21s	-	-
hd-03	5	1m 08s	5	1s	-	-	5	4s	-	-
hd-04	10	9m 06s	9	1s	x1.00	7m 36s	8	10s	x1.04	3m 20s
hd-05	7	6m 08s	7	5s	-	-	7	3m 28s	-	-
hd-06	7	6m 14s	7	4s	-	-	7	2m 30s	-	-
hd-07	5	1m 02s	5	1s	-	-	3	3s	x0.76	24s
hd-08	6	2m 18s	5	1s	x1.00	1m 03s	5	2s	x1.00	1m 00s
hd-09	14	13m 03s	12	2s	x1.10	9m 34s	10	1m 04s	x1.32	7m 56s
hd-10	6	4m 24s	5	1s	x1.03	2m 07s	5	3s	x1.03	1m 25s
hd-11	18	33m 31s	17	2s	x1.00	28m 30s	15	1m 18s	x1.00	21m 40s
hd-12	16	22m 31s	15	1m 35s	x1.00	18m 1s	15	26s	x1.00	17m 04s
bar	12	56m 55s	12	59s	-	-	11	4h 49m 30s	x0.96	48m 19s
cavlc	16	26m 35s	10	1m 48s	x1.20	15m 1s	10	8m 48s	x1.02	13m 06s
ctrl	8	3m 06s	6	3s	x1.02	2m 44s	5	22s	x1.12	1m 18s
dec	3	38s	3	1s	-	-	3	5s	-	-
i2c	15	51m 00s	9	2m 47s	x1.08	21m 38s	8	16m 52s	x1.05	15m 59s
int2float	15	15m 23s	9	10s	x1.13	6m 30s	8	1m 05s	x1.10	4m 09s
router	19	37m 26s	10	14s	x1.31	12m 34s	10	1m 49s	x1.12	12m 31s

speedups for hd-04, hd-07, hd-08 and hd-10 are remarkable (2.7x, 2.6x, 2.3x and 3.1x, respectively). However, both of the two optimizers fail to optimize the other 5 benchmarks, which are relatively simple. Based on the fact that these

small and tricky algorithms are designed to efficiently perform computations on plaintexts, we suspect these benchmarks to be depth-optimal. As of the EPFL benchmarks, 6 out of 7 observe improvements. Both optimizers fail to optimize dec, which we suspect to be already depth-optimal. For bar, we observe 1.18x speedup. For the other benchmarks (cavlc, ctrl, i2c, int2float and router), Lobster achieves remarkable speedups (2.0x – 3.7x). The number of AND gates increases 1.17x more on average. For the 4 sorting benchmarks ({m,i,b,o}sort), we observe nearly 2x increases. For hd09, we observe 1.32x increase. For the other benchmarks, we observe negligible increases. The increases in the number of XOR gates is similar, with the geometric mean of 1.2x. In terms of time spent for the optimization, Lobster successfully optimizes circuits within several minutes in most cases, with the exception of sizeable circuits such as the four sorting benchmarks and bar benchmark that require up to 8 hours.

Learning Capability. We investigate the learned rewrite rules. From all the benchmarks, our rule learner mines 186 rewrite rules. The rule sizes (the size of a rule $l \longrightarrow r$ is measured by $|l|$) range from 4 to 38. The average and median sizes are 12 and 11, respectively. The machine-found optimization patterns are surprisingly aggressive. For example, the following intricate rules enable to reduce the depth of a rewritten critical path by 1 when applied once (we denote $1 \oplus c$ as $\neg c$).

$$(v_1 \wedge (v_2 \wedge ((v_3 \oplus (v_4 \wedge v_5)) \oplus (v_6 \wedge v_5)))) \longrightarrow (((((v_6 \oplus v_4) \wedge v_5) \oplus v_3) \wedge (v_2 \wedge v_1))$$

$$((\neg((v_1 \wedge (\neg(v_2 \oplus v_3))) \oplus (v_2 \oplus v_3))) \wedge v_4) \longrightarrow (((\neg v_2) \oplus v_3) \wedge ((\neg v_1) \wedge v_4))$$

$$(\neg((\neg((((v_1 \oplus v_2) \wedge v_3) \wedge v_4) \wedge v_5)) \oplus v_2)) \longrightarrow ((((v_2 \oplus v_1) \wedge v_4) \wedge (v_3 \wedge v_5)) \oplus v_2)$$

$$((\neg((v_1 \oplus (v_2 \wedge v_3)) \oplus (v_4 \wedge v_3))) \wedge v_5) \longrightarrow (((v_2 \oplus v_4) \wedge (v_5 \wedge v_3)) \oplus ((\neg v_1) \wedge v_5))$$

Next, we investigate how long it takes to learn rewrite rules. The offline learning algorithm (Algo. 1) is time consuming. The timeout limit for the offline learning is set to 168 hours (i.e., 1 week), and we use intermediate results (rules collected so far) when the budget expires. On average, the offline learning phase for each benchmark takes 125 hours. For dsort, hd01, hd02, hd03, hd10, ctrl and dec, the learning takes 1 – 46 hours. For router, it takes 129 hours. The other benchmarks takes 168 hours (i.e., the learner is forced to stop when the time budget expires).

5.3 Comparison to the Baseline

Lobster significantly outperforms the existing state-of-the-art [14] in terms of both optimization time and homomorphic evaluation time. Only 15 out of 25 benchmarks can be optimized by [14], whereas Lobster is able to optimize 19. The maximum speedup of [14] is 3.0x with the geometric mean of 1.58x. The maximum depth reduction ratio is 47.4% with the geometric mean of 15.7%.

We empirically observe that [14] often falls into the basin of local minima because its two rewrite rules can modify only a small area at a time. On the contrary, Lobster often applies large rewrite rules and escapes local optima.

5.4 Efficacy of Reusing Learned Rewrite Rules

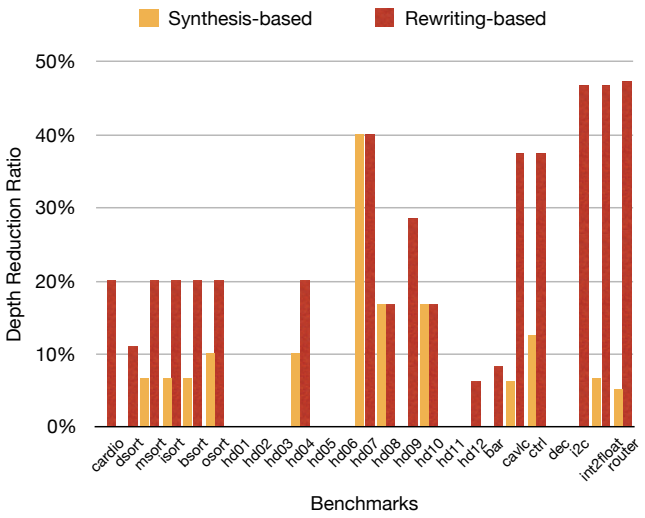

Figure 4. Efficacy of reusing learned rewrite rules

To investigate the benefit of reusing learned rewrite rules, we compare Lobster to a simpler method that uses the offline rule learner as an on-the-fly optimization synthesizer. The timeout limit for optimization is again set to 12 hours, and we use the best intermediate results when the budget expires.

Fig. 4 summarizes the results. The synthesis-based optimizer can optimize only 12 benchmarks within the timeout limit. Furthermore, in all the 12 benchmarks, the depth reduction ratio is less than that of Lobster that reuses learned rewrite rules. That is due to its limited scalability; if the synthesis-based optimizer is given 7 days, it can achieve optimization effects similar to Lobster's. Such enormous optimization costs are mainly due to the inability to prove unrealizability (i.e., no solution) of attempts of optimizing already depth-optimal circuit regions. In such cases, the synthesizer wastes the timeout limit of 1 hour. On the contrary, Lobster can avoid such situations by giving up cases beyond the reach of previously learned rules.

5.5 Efficacy of Equational Rewriting

We now evaluate the effectiveness of design choices made in Lobster– the rule normalization and equational term rewriting. We compare Lobster with its variant without the two techniques. In other words, the variant uses syntactic

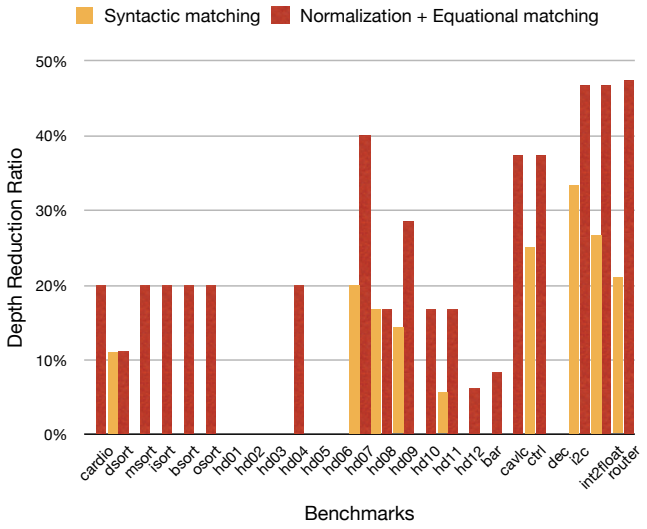

Figure 5. Efficacy of equational rewriting

matching instead of equational matching when conducting term rewriting and applies the learned rules without the normalization process. Fig. 5 summarizes the results. The variant can optimize only 9 benchmarks (Lobster can optimize 19). We conclude that overall, the rule normalization and equational term rewriting play crucial roles in giving flexibility to the rewriting procedure.

5.6 Sensitivity to Changes in a Training Set

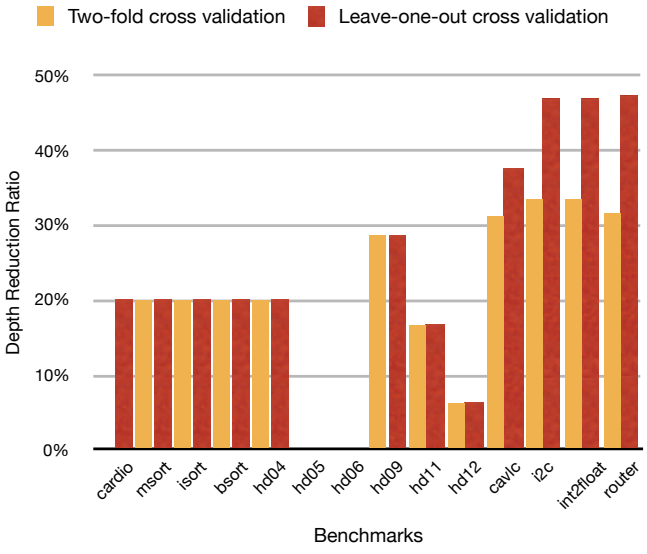

Figure 6. Sensitivity to changes in a training set; comparison of the result of two-fold cross validation with that of leave-one-out cross validation

We now investigate the effects of changing the number of training programs. We have conducted 2-fold cross validation; for each of four benchmark categories (Cingulata, Sorting, HD, EPFL), we used rules learned from the smaller half and applied them to the other larger half, and compare with the result of leave-one-out cross validation. The 14 benchmarks on the x-axis in Fig. 6 are testing benchmarks, and the other 11 benchmarks are training benchmarks. As can be seen in Fig. 6 that summarizes the results, the smaller set of training programs does not lead to significant performance degradation. The cardio, cavlc, i2c, int2float and router benchmarks observe optimization effects less powerful than before, but the other benchmarks remain the same. We conclude that overall, the performance of Lobster is not sensitive to changes in a given set of training programs.

6 Related Work

FHE Compilers. FHE compilers [4, 15, 22, 23] allow programmers to easily write FHE applications without detailed knowledge of the underlying FHE schemes. These compilers also provide optimizations for reducing the multiplicative depth of the compiled circuits. However, the optimization rules used by modern FHE compilers are hand-written, which requires manual effort and is likely to be sub-optimal. In this paper, we aimed to automatically generate optimization rules that can be used by existing compilers.

Cingulata [15] is an open-source compiler that translates high-level programs written in C++ into boolean circuits. Cingulata also supports optimization of circuits for reducing multiplicative depth. It uses ABC [12], an open-source boolean circuit optimizer. Cingulata also uses more advanced, yet hand-written, circuit optimization techniques specially designed for minimizing multiplicative depth [5, 14]. In particular, the multi-start heuristic by Carpov et al. [14], which we used for comparison with Lobster in Section 5, shows a significant reduction in multiplicative depths for their benchmarks. However, we note that the benchmark circuits used in [14] are "intendedly suboptimal to test the ability of optimization tools" [1]. By contrast, the benchmarks used in this paper include circuits that are already carefully optimized in terms of FHE evaluation as explained in Section 5.1, thereby leaving relatively small room for depth reduction. We observe the heuristic in [14] does not perform very well for such a harder optimization task. We recently implemented Aubry et al. [5] and observed that Aubry et al. [5] is slightly better than Carpov et al. [14] (16.9% vs. 15.7% in terms of geometric mean of depth reduction ratio) for our benchmarks.

Ramparts [4] is a compiler for translating programs written in Julia into circuits for homomorphic evaluation. It optimizes the size and multiplicative depth of the circuits using symbolic execution. It also automatically selects the parameters of FHE schemes and the plain text encoding for input values. Ramparts uses a number of hand-written circuit optimization rules for reducing multiplicative depth.

Alchemy [22] is a system that provides domain-specific languages and a compiler for translating high-level plaintext

programs into low-level ones for homomorphic evaluation. The compiler automatically controls the ciphertext size and noise by choosing FHE parameters, generating keys and hints, and scheduling maintenance operations. The domain-specific languages are statically typed and are able to check the safety conditions that parameters should satisfy.

CHET [23] is a domain-specific compiler for FHE applications on neural network inference. It enables a number of optimizations automatically but they are specific to tensor circuits, e.g., determining efficient tensor layout, selecting good encryption parameters, etc. By contrast, our technique is domain-unaware and does not rely on a limited set of hand-written rules.

Superoptimization. Similar to ours, existing superoptimizers [7, 13, 33, 42, 43] for traditional programs are able to learn rewrite rules automatically. The major technical difference, however, is that we use equational matching, rather than syntactic matching, to maximize generalization.

Bansal and Aiken [7] present a technique for automatically constructing peephole optimizers. Given a set of training programs, the technique learns a set of replacement rules (i.e. peephole optimizers) using exhaustive enumeration. The correctness of the learned rules is ensured by a SAT solver. The learned rules are stored in an optimization database and used for other unseen programs via syntactic pattern matching. Optgen [13] is also based on enumeration for generating peephole optimization rules that are sound and complete up to a certain size by generating all rules up to the size and checking the equivalence by an SMT solver. Souper [42] is similar to [7] but is based on a constraint-based synthesis technique and targets a subset of LLVM IR. STOKE [33, 43] uses a stochastic search based on MCMC to explore the space of all possible program transformations for the x86-64 instruction set.

Program Synthesis. Over the last few years, inductive program synthesis has been widely used in various application domains (e.g. [26–29, 45, 51, 53]). In this work, we use inductive synthesis to minimize multiplicative depth of boolean circuits. To our knowledge, this is the first application of program synthesis for efficient homomorphic evaluation. Our work has been inspired by the prior work by Eldib et al. [26], where syntax-guided synthesis and static analysis are used to automatically transform a circuit into an equivalent and provably secure one that is resistant to a side-channel attack.

Term Rewriting. Term rewriting has been widely used in program transformation systems (e.g. [9, 11, 47, 48, 50]). The previous rewrite techniques rely on hand-written rules that require domain expertise, whereas this work uses automatically synthesized rewrite rules. For example, Chiba et al. [20] presented a framework of applying code-transforming templates based on term rewriting, where programs are represented by term rewriting systems and transformed by a set of given rewrite rules (called templates). Visser et al. [50] used term rewriting in ML compilers and presented a language for writing rewriting strategies. In this work, we focus on a different application domain of term rewriting (i.e. homomorphic evaluation) and provide a novel idea of learning and using rewrite rules automatically.

7 Conclusion

In this paper, we presented a new method for optimizing FHE boolean circuits that does not require any domain expertise and manual effort. Our method first uses program synthesis to automatically discover a set of optimization rules from training circuits. Then, it performs term rewriting on the new, unseen circuit based on the equational matching to maximally leveraging the learned rules. We demonstrated the effectiveness of our method with 18 FHE applications from diverse domains. The results show that our method achieves sizeable optimizations that are complementary to existing domain-specific optimization techniques.

Acknowledgments

This work was partially supported by Korea Institute for Information & Communications Technology Promotion (No. 2017-0-00616), National Research Foundation of Korea (No. 2019R1G1A1100293, No. 2020R1C1C1014518, No. 21A201511-13068, & No. 2017M3C4A7068175), Samsung Electronics (No. SRFC-IT1502-53), SK Hynix (No. 0536-20190093) and Hanyang University (No. HY-2018).

References

[1] 2015. The EPFL Combinational Benchmark Suite. https://www.epfl.ch/labs/lsi/page-102566-en-html/benchmarks/.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design (FMCAD '13)*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336.

[4] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '19)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/3338469.3358945

[5] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. 2019. Faster homomorphic encryption is not enough: improved heuristic for multiplicative depth minimization of Boolean circuits. Cryptology ePrint Archive, Report 2019/963. https://eprint.iacr.org/2019/963.

[6] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA.

[7] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*. ACM, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

[8] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J. Wu. 2013. Private Database Queries Using Somewhat Homomorphic Encryption. In *Applied Cryptography and Network Security*, Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–118.

[9] James M. Boyle, Terence J. Harmer, and Victor L. Winter. 1997. Modern Software Tools for Scientific Computing. Birkhauser Boston Inc., Cambridge, MA, USA, Chapter The TAMPR Program Transformation System: Simplifying the Development of Numerical Software, 353–372. http://dl.acm.org/citation.cfm?id=266469.266509

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. ACM, New York, NY, USA, 309–325. https://doi.org/10.1145/2090236.2090262

[11] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1 (2008), 52 – 70. https://doi.org/10.1016/j.scico.2007.11.003 Special Issue on Second issue of experimental software and toolkits (EST).

[12] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-strength Verification Tool. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV '10)*. Springer-Verlag, Berlin, Heidelberg, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

[13] Sebastian Buchwald. 2015. Optgen: A Generator for Local Optimizations. In *Compiler Construction*, Björn Franke (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–189.

[14] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. 2018. A Multi-start Heuristic for Multiplicative Depth Minimization of Boolean Circuits. In *Combinatorial Algorithms*, Ljiljana Brankovic, Joe Ryan, and William F. Smyth (Eds.). Springer International Publishing, Cham, 275–286.

[15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing (SCC '15)*. ACM, New York, NY, USA, 13–19. https://doi.org/10.1145/2732516.2732520

[16] S. Carpov, T. H. Nguyen, R. Sirdey, G. Constantino, and F. Martinelli. 2016. Practical Privacy-Preserving Medical Diagnosis Using Homomorphic Encryption. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD '16)*. 593–599. https://doi.org/10.1109/CLOUD.2016.0084

[17] Gizem S. Cetin, Yarkin Doroz, Berk Sunar, and Erkay Savas. 2015. Depth Optimized Efficient Homomorphic Sorting. In *Proceedings of the 4th International Conference on Progress in Cryptology - Volume 9230 (LATINCRYPT '15)*. Springer-Verlag, Berlin, Heidelberg, 61–80. https://doi.org/10.1007/978-3-319-22174-8_4

[18] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology (ASIACRYPT '17)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[19] Jung Hee Cheon, Miran Kim, and Kristin Lauter. 2015. Homomorphic Computation of Edit Distance. In *Financial Cryptography and Data Security*, Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–212.

[20] Yuki Chiba, Takahito Aoto, and Yoshihito Toyama. 2005. Program Transformation by Templates Based on Term Rewriting. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05)*. ACM, New York, NY, USA, 59–69. https://doi.org/10.1145/1069774.1069780

[21] Cingulata 2019. Cingulata. https://github.com/CEA-LIST/Cingulata. CEA-LIST.

[22] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easY. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1020–1037. https://doi.org/10.1145/3243734.3243828

[23] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-homomorphic Neural-network Inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, New York, NY, USA, 142–156. https://doi.org/10.1145/3314221.3314628

[24] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully Homomorphic Encryption over the Integers. In *EUROCRYPT 2010*.

[25] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. 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 - Volume 48 (ICML '16)*. JMLR.org, 201–210. http://dl.acm.org/citation.cfm?id=3045390.3045413

[26] Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 343–363.

[27] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[28] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

[29] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[30] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC '09)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[31] HEAAN 2019. HEAAN. https://github.com/snucrypto/HEAAN. SNU Crypto Group.

[32] HElib 2019. HElib. http://github.com/homenc/HElib. IBM Research.

[33] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[34] Nick Howgrave-Graham. 2001. Approximate Integer Common Divisors. In *CaLC*.

[35] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[36] Deepak Kapur and Paliath Narendran. 1987. Matching, Unification and Complexity. *SIGSAM Bull.* 21, 4 (Nov. 1987), 6–9. https://doi.org/

10.1145/36330.36332

[37] Woosuk Lee, Hyunsook Hong, Kwangkeun Yi, and Jung Hee Cheon. 2015. Static Analysis with Set-Closure in Secrecy. In *Static Analysis*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–35.

[38] Wenjie Lu, Shohei Kawasaki, and Jun Sakuma. 2016. Using Fully Homomorphic Encryption for Statistical Analysis of Categorical, Ordinal and Numerical Data. *IACR Cryptology ePrint Archive* 2016 (2016), 1163.

[39] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. 2011. Can Homomorphic Encryption Be Practical?. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2046660.2046682

[40] Goldreich Oded. 2009. *Foundations of Cryptography: Volume 2, Basic Applications* (1st ed.). Cambridge University Press, New York, NY, USA.

[41] Detlef Plump. 2002. Essentials of Term Graph Rewriting. *Electronic Notes in Theoretical Computer Science* 51 (2002), 277 – 289. https://doi.org/10.1016/S1571-0661(04)80210-X GETGRATS Closing Workshop.

[42] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422 (2017). arXiv:1711.04422 http://arxiv.org/abs/1711.04422

[43] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[44] SEAL 2019. Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[45] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

[46] SyGuS 2019. The 6th Syntax-Guided Synthesis Competition. https://sygus.org/comp/2019/. SyGuS-Comp 2019.

[47] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[48] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. 2002. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Trans. Program. Lang. Syst.* 24, 4 (July 2002), 334–368. https://doi.org/10.1145/567097.567099

[49] Alexander Viand and Hossein Shafagh. 2018. Marble: Making Fully Homomorphic Encryption Accessible to All. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '18)*. ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/3267973.3267978

[50] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/289423.289425

[51] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[52] Henry S. Warren. 2012. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional.

[53] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887