Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-Bounded Exhaustive Search

DONGKWON LEE, Seoul National University
WOOSUK LEE, Hanyang University
HAKJOO OH, Korea University
KWANGKEUN YI, Seoul National University

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, term rewriting, and equality saturation. 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 uses the equational matching with generalized version of the learned rules, and its soundness property is formally proven. Our optimizations also try to explore every possible alternative order of applying rewrite rules by time-bounded exhaustive search technique called equality saturation. Experimental results show that our method generates circuits that can be homomorphically evaluated 1.08x – 3.17x faster (with the geometric mean of 1.56x) than the state-of-the-art method. Our method is also orthogonal to existing domain-specific optimizations.

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

Additional Key Words and Phrases: Homomorphic Encryption Circuit, Program Synthesis, Term Rewriting, Equality Saturation

ACM Reference format:
DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2023. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-Bounded Exhaustive Search. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 111 (August 2023), 38 pages.
https://doi.org/doi

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) [32] 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 about the data

0164-0925/2023/8-ART111 $15.00
https://doi.org/doi

itself, the clients can safely upload their encrypted data without trusting the software/hardware vendors of the cloud.

Problem

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 [33, 34, 51], 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 key challenge is therefore reducing the costs of FHE applications so that they become 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 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 [25], data movement [42] and batching [44], and heuristics for encryption parameter selection [25, 27] enable several orders of magnitude speedups in a wide range of FHE applications (such as image recognition [25], statistical analysis [42], sorting [17], bioinformatics [19], database [8], and static program analysis [41]). Yet, such improvement requires 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 plain-text programs into optimized FHE code. For example, Ramparts [4], Cingulata [15] and Alchemy [23] take programs written in a high-level language (e.g., Julia, C++, or a custom DSL) and transform them into arithmetic 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, 42] to achieve the desired efficiency. Furthermore, state-of-the-art compilers rely on hand-written optimization rules whose findings require expertise and are likely to remain sub-optimal.

Our Approach

In this paper, in the context of optimizing FHE compilation, 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 manually 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, 56], 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.

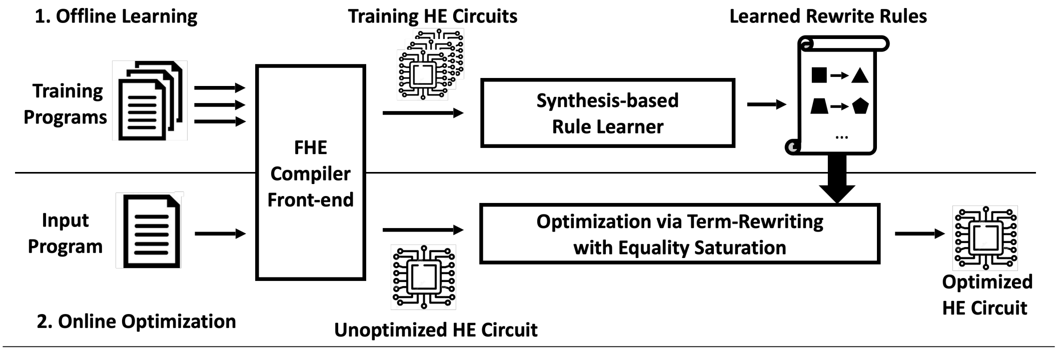

Fig. 1. Overview of the system.

To achieve this critical optimization for homomorphic computation circuits as much as possible, we combine three techniques: program synthesis, term rewriting and equality saturation. Fig. 1 illustrates 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. Most of these machine-found optimization patterns are what we can hardly imagine from human-devised optimization techniques. (Section 5.2)
- 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 generalize what have been learned from training circuits by giving flexibility to the rewriting procedure: our method is based on a limited version of the equational matching that takes commutativity into account rather than just the syntactic matching. Our rewriting method is proven to be sound and terminating.
- Moreover, by the equality saturation technique [54], we explore every possible alternative order of applying rewrite rules (i.e. we obtain backtracking effect). First, we efficiently express all possible result circuits as a form of program grammar using a data structure called E-graph [61] (saturation process). Next, from the saturated E-graph, we extract the optimal circuit that has the lowest multiplicative depth (extraction process).

We implement our method atop Cingulata [21], a widely-used FHE compiler and evaluate our method on 25 FHE applications from diverse domains (statistical analysis, sorting, medical diagnosis, low-level algorithms, etc). 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 1.08x – 3.17x faster (with the geometric mean of 1.56x) 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 with the equational matching for generalized versions of the learned rewrite rules on a given new circuit. The soundness property of this rewriting system are

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

formally proven. We combine equality saturation [54] with the existing term rewriting system (Section 4.4) to obtain a backtracking effect. This saturation-based term rewriting system outperforms our previous approach [39].

- 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:

$$\begin{array}{ll} \mathsf{Enc}_{pk} : \mathbb{Z}_2 \to \Omega & \mathsf{Dec}_{sk} : \Omega \to \mathbb{Z}_2 \\ \mathsf{Add}_{pk} : \Omega \times \Omega \to \Omega & \mathsf{Mul}_{pk} : \Omega \times \Omega \to \Omega \end{array}$$

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 [26] based on approximate common divisor problems [36]:

- The secret key (sk) is a random integer p.
- For a plaintext m, 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 [47].
- For a ciphertext $\bar{c}$, Dec($\bar{c}$) outputs (($\bar{c}$ mod p) mod 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 $\mathrm{Dec}(\bar{c}_i) = ((\bar{c}_i \bmod p) \bmod 2) = m_i$, if $|2r_i + m_i| < p/2$. Then, the following equations hold:

$$\begin{aligned} \bar{c}_1 + \bar{c}_2 &= p(q_1 + q_2) + \underbrace{2(r_1 + r_2) + m_1 + m_2}_{\text{noise}_{\mathsf{Add}}} \\ \bar{c}_1 \times \bar{c}_2 &= p(pq_1q_2 + \cdots) + \underbrace{2(2r_1r_2 + r_1m_2 + r_2m_1) + m_1m_2}_{\text{noise}_{\mathsf{Mult}}} \end{aligned}$$

Based on these properties, we can show that

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

if the absolute values of $\text{noise}_{\text{Add}}$ and $\text{noise}_{\text{Mult}}$ 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 $\ell : \mathbb{C} \longrightarrow \mathbb{N}$ be a function that computes the multiplicative depth of a circuit, which is inductively defined as follows:

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

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:

- If $c = x$ or 0 or 1, $\mathcal{P}(c) \overset{\text{def}}{=} \{\epsilon\}$, where ϵ is the empty string.
- If $c = f(c_1, c_2)$ where $f \in \{\wedge, \oplus\}$, then

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

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

Example 2.1. Consider a circuit $c(v_1, v_2, v_3, v_4)$ defined as

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

[2]A leveled scheme may be turned into a fully homomorphic one by introducing a bootstrapping operation [32], which is computationally heavy.

The multiplicative depth $\ell(c)$ of circuit 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 $C\mathcal{P}(c)$ 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 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 multiplicative depth is smaller than c. Formally, our goal is to find c' such that

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

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

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 2.1 in the training set:

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

The depth of this circuit is 3 and we would like to find a semantically-equivalent circuit c' with a smaller depth (i.e. $\ell(c') \leq 2$). 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:

$$\begin{array}{rcl}
S & \longrightarrow & d_3 \\
d_3 & \longrightarrow & d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 & \longrightarrow & d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 & \longrightarrow & d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 & \longrightarrow & 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{array}$$

where S denotes the start symbol, and each non-terminal symbol d_i denotes circuits of multiplicative depth $\leq i$. 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. EUSolver [3], DUET [40]) is able to find the following circuit c':

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

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, $\neg(v_2 \land v_3)$ in c and $\neg(v_3 \land v_2)$ in c' are equivalent modulo commutativity and therefore we replace them by a new variable x, which simplifies c and c' into $v_1 \land (\neg(v_4 \land x))$ and $(x \land (v_1 \land v_4)) \oplus v_1$, 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:

$$v_1 \land (\neg(v_4 \land x)) \rightarrow (x \land (v_1 \land v_4)) \oplus v_1. \tag{2}$$

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:

$$((v_5 \land v_6) \land (\neg((v_7 \land v_8) \land (\neg((v_8 \land v_9) \land (v_9 \land v_7)))))). \tag{3}$$

To optimize the circuit, we first compare it with the left-hand side of the learned rewrite rule (i.e. $v_1 \land (\neg(v_4 \land x))$), 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:

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

Note that $\sigma(v_1 \land (\neg(v_4 \land x)))$ 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:

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

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

As described from the above, we obtain rewrite rules from a small circuit and apply it into a new 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

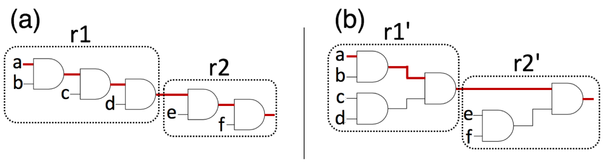

Fig. 2. (a) The circuit c_{ex} of depth 5. (b) A circuit that has depth 3 and the same semantics as c_{ex}.

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):

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

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

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

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:

$$r_1' \overset{\text{def}}{=} (a \land b) \land (c \land d)$$

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

$$r_2' \overset{\text{def}}{=} r_1' \land (e \land f)$$

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.

Backtracking via Equality Saturation

In rewriting input circuits, we may miss the global optimality because there can be multiple targets to optimize and multiple rewrite rules to apply to each target. In this situation, optimization results can be varied depending on which target is rewritten first, or which rewrite rule is applied first.

For example, suppose that we want to optimize circuit $c_0 = ((x_1 \land x_2) \land (x_2 \oplus x_3)) \land x_3$ using the following learned rewrite rules.

$$\begin{array}{lrcl}
\text{rule (1)}: & ((v_1 \land v_2) \land v_3) \land v_4 & \longrightarrow & ((v_1 \land v_2) \land v_4) \land ((v_2 \oplus v_4) \oplus v_3) \\
\text{rule (2)}: & ((v_1 \land v_2) \land v_3) \land v_4 & \longrightarrow & (v_1 \land v_2) \land (v_3 \land v_4) \\
\text{rule (3)}: & (v_1 \oplus v_1) & \longrightarrow & 0 \\
\text{rule (4)}: & (v_1 \land 0) & \longrightarrow & 0
\end{array}$$

We can apply both rule (1) and rule (2) using substitution $\sigma = \{v_1 \mapsto x_1, v_2 \mapsto x_2, v_3 \mapsto (x_2 \oplus x_3), v_4 \mapsto x_3\}$. If we apply rule (1) first, we can optimize $c_0 \longrightarrow c_1 = ((x_1 \land x_2) \land x_3) \land ((x_2 \oplus x_3) \oplus (x_2 \oplus x_3))$. In this case, we can subsequently apply rule (3) and rule (4) and optimize $c_0 \longrightarrow^* 0$. Otherwise, we can optimize $c_0 \longrightarrow c_2 = (x_1 \land x_2) \land ((x_2 \oplus x_3) \land x_3)$. In this case, we can no longer apply the rewrite rules and miss the opportunity of further optimizing c_0.

To address this rewrite order issue, we use the equality saturation technique [54] that obtains a fully backtracking effect within a given time limit. By the technique we try to explore every possible rewriting sequences (e.g. we try to explore both $c_0 \longrightarrow^* 0$ and $c_0 \longrightarrow c_2$).

Equality saturation consists of two processes: saturation process and extraction process. First, in the saturation process, we express all possible result circuits as a form of program grammar using a data structure named E-graph [61] (e.g. we construct program grammar that can generate circuits c_0, c_1, c_2 and 0). In the extraction process, we extract an optimal circuit that has the lowest multiplicative depth from a saturated E-graph (e.g. we extract circuit 0 which has the lowest multiplicative depth). Details of this equality saturation process are in Section. 4.4.

4 ALGORITHM

We first review (Section 4.1) key definitions and results borrowed from Baader and Nipkow [6] that will be used in the rest of the paper. Then we present the offline learning phase (Section 4.2), online optimization phase (Section 4.3) based on term rewriting and backtracking system(Section 4.4) via equality saturation.

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

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) $s\ |_\epsilon \overset{\text{def}}{=} s$ and (ii) $f(s_1, \cdots, s_n)\ |_{iq} \overset{\text{def}}{=} s_i\ |_q$. 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,

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

Substitution

A $T_{\Sigma,X}$-substitution is a function $X \longrightarrow T_{\Sigma,X}$. 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 $\hat{\sigma} : T_{\Sigma,X} \longrightarrow T_{\Sigma,X}$ as follows: for $x \in X$, $\hat{\sigma}(x) \overset{\text{def}}{=} \sigma(x)$ and for any non-variable term $s = f(s_1, \cdots, s_n)$, $\hat{\sigma}(s) \overset{\text{def}}{=} f(\hat{\sigma}(s_1), \cdots, \hat{\sigma}(s_n))$. 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 $l \longrightarrow r$, 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 $\longrightarrow_E$ on $T_{\Sigma,X}$ induced by a term rewriting system E is defined as follows:

$$\begin{array}{lcl} s \longrightarrow_E t & \iff & \exists l \rightarrow r \in E, p \in Pos(s), \sigma \in Sub(T_{\Sigma,X}). \\ & & s\,|_p = \sigma(l), t = s[p \leftarrow \sigma(r)] \end{array}$$

Example 4.1. Let $E = \{x \wedge (y \wedge z) \approx (x \wedge y) \wedge z, 1 \wedge x \approx x, x \wedge y \approx y \wedge x\}$. Then, $1 \wedge (a \wedge 1) \longrightarrow_E (1 \wedge a) \wedge 1 \longrightarrow_E a \wedge 1 \longrightarrow_E 1 \wedge a \longrightarrow_E a$.

Equational Theory

Let $\leftrightarrow_E^*$ denote the reflexive-transitive-symmetric closure of $\longrightarrow_E$. 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 $\approx_E \overset{\text{def}}{=} \{\langle s, t \rangle \in T_{\Sigma,X} \times T_{\Sigma,X} \mid E \models s \approx t\}$ is called the equational theory induced by E.

Example 4.2. Let $C = \{x \wedge y \approx y \wedge x, x \oplus y \approx y \oplus x\}$. Then, $x \wedge (y \oplus z) \longrightarrow_C (y \oplus z) \wedge x \longrightarrow_C (z \oplus y) \wedge x$. The theory of commutativity for circuits is $\approx_C \overset{\text{def}}{=} \{\langle s, t \rangle \in \mathbb{C} \times \mathbb{C} \mid C \models s \approx t\}$ (e.g. $x \wedge (y \oplus z) \approx_C (z \oplus y) \wedge x$).

Example 4.3. Boolean ring theory is $\approx_{\mathcal{R}} \overset{\text{def}}{=} \{\langle s, t \rangle \in \mathbb{C} \times \mathbb{C} \mid \mathcal{R} \models s \approx t\}$ where

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

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 [6].

Example 4.4. The original circuit c and its optimized version c' in Section 3 are semantically equivalent because

$$\begin{array}{ll} c = v_1 \wedge (1 \oplus (v_4 \wedge (\neg(v_2 \wedge v_3)))) & \\ \longrightarrow_{\mathcal{R}} (v_1 \wedge 1) \oplus (v_1 \wedge (v_4 \wedge (\neg(v_2 \wedge v_3)))) & x \wedge (y \oplus z) \approx (x \wedge y) \oplus (x \wedge z) \\ \longrightarrow_{\mathcal{R}} v_1 \oplus (v_1 \wedge (v_4 \wedge (\neg(v_2 \wedge v_3)))) & x \wedge y \approx y \wedge x, 1 \wedge x \approx x \\ \longrightarrow_{\mathcal{R}} v_1 \oplus ((v_1 \wedge v_4) \wedge (\neg(v_3 \wedge v_2))) & (x \wedge y) \wedge z \approx x \wedge (y \wedge z) \\ \longrightarrow_{\mathcal{R}} ((v_1 \wedge v_4) \wedge (\neg(v_3 \wedge v_2))) \oplus v_1 & x \oplus y \approx y \oplus x \\ \longrightarrow_{\mathcal{R}} ((\neg(v_3 \wedge v_2)) \wedge (v_1 \wedge v_4)) \oplus v_1 = c' & x \wedge y \approx y \wedge x \end{array}$$

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 4.5. 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 [28], which uses syntax-guided synthesis to automatically transform a circuit into an equivalent and provably secure one.

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.

Algorithm 1 Synthesis-based Rule Learning

```
Input: $c$: input boolean circuit
Input: $\theta$: threshold for termination condition
Input: $n$: predefined size limit for chosen regions
Output: $E$: a set of rewrite rules
 1: $c' \leftarrow c$
 2: $E \leftarrow \emptyset$
 3: $w \leftarrow \mathcal{CP}(c')$
 4: while $w \neq \emptyset$ and $\frac{|c'|}{|c|} < \theta$ do
 5:     remove a $pos$ from $w$
 6:     $\langle r, \sigma \rangle \leftarrow$ GetRegion($c' |_{pos}, n$)
 7:     $r' \leftarrow$ Synthesize($r, \ell(r) - 1, \sigma$)
 8:     if $r' \neq \bot$ then
 9:         $E \leftarrow E \cup \{$Normalize($r \rightarrow r'$)$\}$
10:         $c' \leftarrow c'[pos \leftarrow \sigma(r')]$
11:         $w \leftarrow \mathcal{CP}(c')$
12:     end if
13: end while
14: return $E$
```

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 $|r| \leq n$ (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 $r \longrightarrow r'$. We generalize the rule by invoking the Normalize procedure (Section 4.2.4), 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 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.)).

Algorithm 2 GetRegion

```
Input: $c$: input boolean circuit region
Input: $n$: predefined size limit for regions
Output: $r$: a circuit region
Output: $\sigma$: 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 $\langle c[1 \leftarrow r', 2 \leftarrow x], \sigma \cup \{x \mapsto c_2\}\rangle$
10: else
11:     $\langle r', \sigma\rangle \leftarrow$ GetRegion($c_2, n-1$)
12:     return $\langle c[1 \leftarrow x, 2 \leftarrow r'], \sigma \cup \{x \mapsto c_1\}\rangle$
13: end if
```

Example 4.6. 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 the 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 4.7. After selecting the region r_2 as in Example 4.6, 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. EUSolver [3], DUET [40]) 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 $l' = \sigma(l)$, $r' = \sigma(r)$, $\sigma = \{\forall l\ |_{p_i} \mapsto x, \forall r\ |_{p_j} \mapsto x\}\ s.t.\ l\ |_{p_i} \approx_C l\ |_{p_l}, r\ |_{p_j} \approx_C r\ |_{p_r}$, 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 4.8. Suppose we want to normalize a rewrite rule:

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

Note that $l\ |_{11} = l\ |_2 = (a \wedge b) \approx_C r\ |_1 = l\ |_{12} = (b \wedge a)$. If we replace the subterms $l\ |_{11}$, $l\ |_{12}$, $l\ |_2$, and $r\ |_1$ with a fresh variable x, we obtain normalized rewrite rule $(x \wedge x) \wedge x \rightarrow x$, which is semantics preserving.

Example 4.9. 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 Optimization without Backtracking

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 $\to_{E,\ell}$ induced by E (learned rewrite rules) and ℓ (the function computing the multiplicative depth).

$$\begin{aligned} s \to_{E,\ell} t \quad \iff \quad & \exists l \to r \in E, p \in \mathcal{CP}(s), \sigma \in \mathit{Sub}(\mathbb{C}). \\ & s\,|_p \approx_C \sigma(l), \ell(\sigma(l)) > \ell(\sigma(r)), t = s[p \leftarrow \sigma(r)]. \end{aligned}$$

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 $s\,|_p \approx_C \sigma(l)$. Here, a complication arises that there may be multiple C-matchers. In such a case, we choose the one that can reduce depth.

Example 4.10. Recall the rewrite rule (2) in Section 3

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

and the target circuit (3) of depth 4

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

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

$$\begin{aligned} \sigma_1(r) &= (\neg((v_8 \land v_9) \land (v_9 \land v_7)) \land ((v_5 \land v_6) \land (v_7 \land v_8))) \oplus (v_5 \land v_6), \\ \sigma_2(r) &= ((v_7 \land v_8) \land ((v_5 \land v_6) \land (\neg(v_8 \land v_9) \land (v_9 \land v_7)))) \oplus (v_5 \land v_6). \end{aligned}$$

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 theorem ensures that our term rewriting system is semantics-preserving and terminating.

Theorem 4.11 (Soundness). $\forall c, c' \in \mathbb{C}.\ c \to_{E,\ell} c' \Rightarrow c \approx_{\mathcal{R}} c'$.

Proof. Available in appendix. □

Theorem 4.12 (Termination). *$\to_{E,\ell}$ is a terminating relation.*

Proof. Available in appendix. □

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 $\to_{E,\ell}$, given a circuit c, we perform term rewriting on c to obtain an optimized circuit c' such that $c \overset{*}{\to}_{E,\ell} c'$. 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 to 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 $l \longrightarrow r$ 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 [38]. We limit the search space of C-matching algorithm by limiting the number of applications of commutativity rules (see appendix 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 [48] 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 Plump [48] for more details about the soundness proof and the translation method.

4.4 Optimization with Backtracking Based on Equality Saturation

4.4.1 E-graph Structure. E-graph structure is defined as a triple of a set of enodes, a set of eclasses, and a set of edges. Each enode contains a boolean operator ($\wedge$, $\oplus$) or boolean value (0, 1, x). Eclass is a set of enodes. Edge connects an enode to an eclass.

The meaning of the E-graph is as follows. Each enode represents a set of all expressions that can be generated in the following manner, and each eclass represents a set of all expressions that can be generated by enodes inside it. All E-graphs must have the invariant that all expressions generated by enodes in the same eclass must be semantically equivalent. If an enode is a boolean value, it generates the constant expression itself. If an enode is a boolean operator, it generates all expressions that can be made by choosing each child boolean expression among the expressions that the corresponding child eclass represents.

Figure 3 shows the concept of an E-graph. Let us illustrate how to generate various expressions from eclass or enode, and show that the above invariant of E-graph holds. Let EC_i and EN_i be the sets of expressions that can be generated by eclass ec_i and enode i respectively. Since enode 1–3 respectively contain a boolean value x_1–x_3, $EC_i = EN_i = \{x_i\}(1 \leq i \leq 3)$. Since enode 4 contains a boolean operator $\wedge$ and has two children eclasses ec_1 and ec_2, it can generate $x_1 \wedge x_2$ by choosing $x_1 \in EC_1$ and $x_2 \in EC_2$ as two children expressions. Similarly, we get $EC_5 = EN_5 = \{(x_1 \wedge x_2) \wedge x_3\}$,

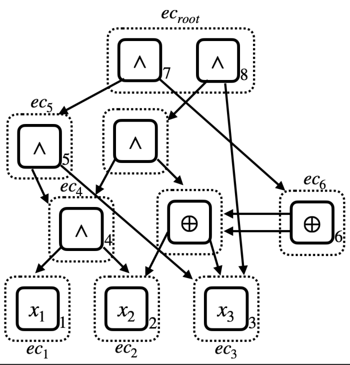

Fig. 3. Simple example of E-graph. Each box means enode, and dotted box means eclass.

$EC_6 = EN_6 = \{(x_2 \oplus x_3) \oplus (x_2 \oplus x_3)\}$, and $EN_7 = \{((x_1 \wedge x_2) \wedge x_3) \wedge ((x_2 \oplus x_3) \oplus (x_2 \oplus x_3))\}$. In the same way, we get $EN_8 = \{((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3\}$. Since ec_{root} contains enodes 7 and 8, we get $EC_{root} = EN_7 \cup EN_8 = \{((x_1 \wedge x_2) \wedge x_3) \wedge ((x_2 \oplus x_3) \oplus (x_2 \oplus x_3)),\ ((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3\}$. Recall that two boolean expressions in EC_{root} are semantically equivalent.

Note that E-graph also can be interpreted as a program grammar. Each of the eclasses corresponds to a nonterminal symbol and each of enodes in that eclass corresponds to a production rule for the nonterminal symbol. Every E-graph corresponds to a particular context-free grammar.

In this context, equality saturation is a process of constructing a program grammar that can generate all boolean circuits equivalent to the input circuit.

4.4.2 Equality Saturation Process. Equality saturation consists of two processes: saturation process and extraction process. First, in the saturation process, we continually expand E-graph by finding all circuits which are semantically equivalent to an initial circuit. We call the E-graph saturated if we can not find any other equivalent circuits. In the extraction process, we extract a circuit that has the lowest multiplicative depth from a saturated E-graph.

Figure 4 and Figure 5 illustrates the saturation process. Same as Section 3, we start with a term graph of the input boolean circuit $((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3$ and following rewrite rules.

$$\begin{array}{lrcl}
\text{rule (1)}: & ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 & \rightarrow & ((v_1 \wedge v_2) \wedge v_4) \wedge ((v_2 \oplus v_4) \oplus v_3) \\
\text{rule (2)}: & ((v_1 \wedge v_2) \wedge v_3) \wedge v_4 & \rightarrow & (v_1 \wedge v_2) \wedge (v_3 \wedge v_4) \\
\text{rule (3)}: & (v_1 \oplus v_1) & \rightarrow & 0 \\
\text{rule (4)}: & (v_1 \wedge 0) & \rightarrow & 0
\end{array}$$

Then we explore optimized circuits that can be generated based on the rewrite rules by an iteration of three steps: ematch, add, and merge. Figure 4 illustrates the first iteration. In the ematch step, we find all enodes that can be rewritten by a certain rule. An enode is said rewritable by a certain rule if it can generate a circuit which can be rewritten by the rule. As the root enode can generate circuit $((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3$ that can be rewritten by rule (1) and rule (2), we get two rewritten circuits as a result of ematch step for the root enode (Figure 4.(a)). In the add step, we add all rewritten circuits to the E-graph in a recursive manner from bottom to top (Figure 4.(b)). Newly added enodes are colored gray in Figure 4 and Figure 5. Note that each added rewritten circuit corresponds to an enode. In the merge step, for each pair of rewritable enode and rewritten circuit, we merge the rewritable enode and newly added enode (that corresponds to the rewritten circuit) as the same eclass (Figure 4.(c)). In the second and third iteration, we expand E-graph by applying rewrite rule

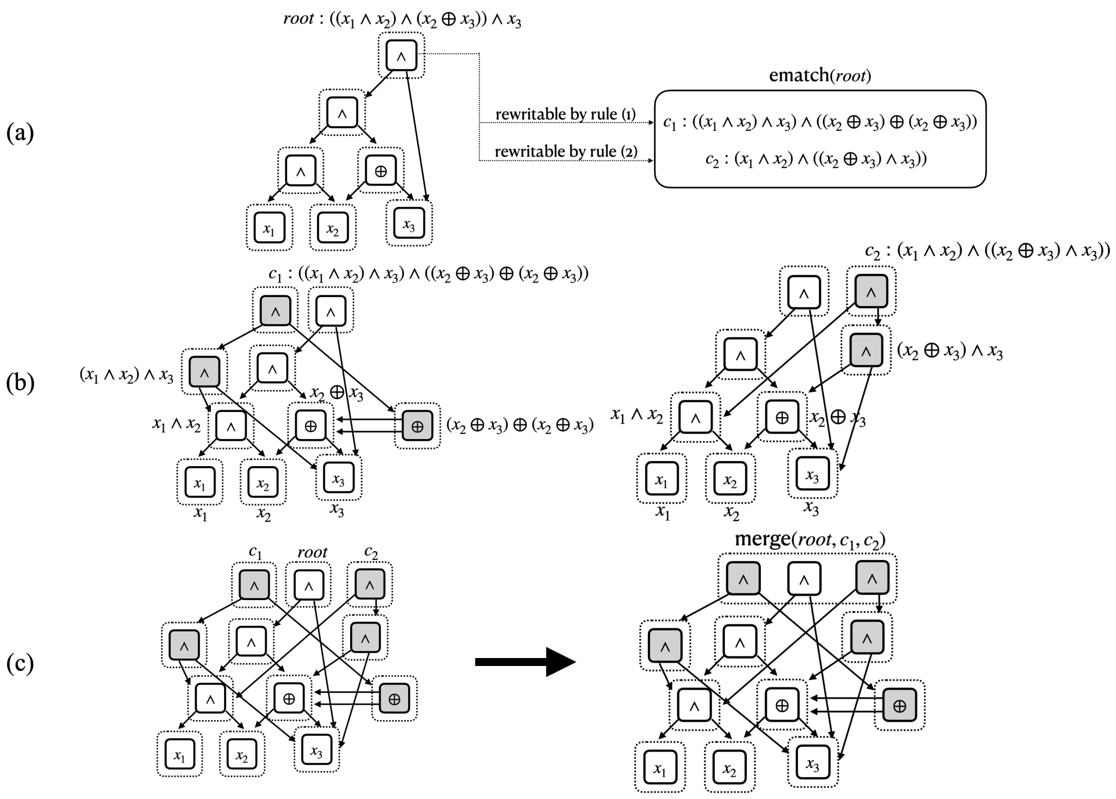
(a)
root : $((x_1 \wedge x_2) \wedge (x_2 \oplus x_3)) \wedge x_3$
ematch(root)
rewritable by rule (1)
rewritable by rule (2)
$c_1 : ((x_1 \wedge x_2) \wedge x_3) \wedge ((x_2 \oplus x_3) \oplus (x_2 \oplus x_3))$
$c_2 : (x_1 \wedge x_2) \wedge ((x_2 \oplus x_3) \wedge x_3))$
(b)
$(x_1 \wedge x_2) \wedge x_3$
$x_2 \oplus x_3$
$x_1 \wedge x_2$
$(x_2 \oplus x_3) \oplus (x_2 \oplus x_3)$
$(x_2 \oplus x_3) \wedge x_3$
(c)
c_1 root c_2
merge(root, c_1, c_2)

Fig. 4. Change of E-graph during a single iteration. Dotted box means eclass. (a) ematch result for root enode. (b) add subcircuit c_1 and c_2 to E-graph. (c) merge root node and result enodes (c_1 and c_2) of add step.

(3) and rule (4) respectively (Figure 5.(c), Figure 5.(d)).[3] As the root enode is merged with the enode that contains 0, we can figure out that the initial circuit is semantically equivalent to 0. More details of these three steps are described in Willsey et al. [61].

We repeat the above three steps until no further changes are made (i.e. E-graph is saturated). Since it is not guaranteed that an E-graph will end up saturating, we give an appropriate amount of time limit (12 hours).

In the extraction process, we extract the least-cost circuit from the saturated E-graph for the given cost model. If the cost function is local (the cost of a node is computable only with the costs of its children nodes), it is known that the least-cost circuit for that cost model can be easily extracted from the E-graph [61]. In our case, since the multiplicative depth of the circuit is a local function, circuits with the lowest multiplication depth can be easily extracted.

4.4.3 Tradeoff between Optimality and Cost. In our single-path (i.e. without backtracking) term rewriting system defined in Section 4.3, we can only explore limited optimization space due to termination property and efficiency. To ensure termination, we selectively rewrite a target circuit only when its multiplicative depth is reduced. For efficiency, we apply rewrite rules only to a target circuit lying on critical paths. For these reasons, we have to give up the guarantee to find a globally optimal circuit for efficient and terminating rewriting procedures.

[3]More enodes are rewritable by rule (2), but we ruled out them in Figure 5 for clarity.

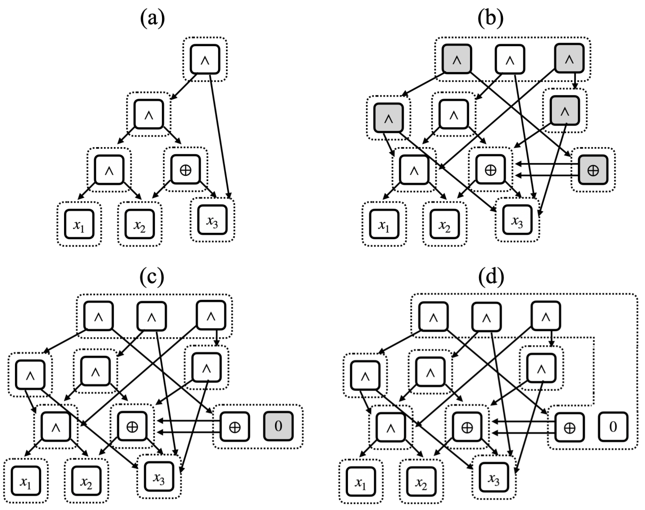

Fig. 5. Change of E-graph during iterations. Dotted box means eclass. (a) initial E-graph. (b) after 1 iteration. (c) after 2 iterations. (d) saturated E-graph.

In a saturation-based rewriting system, the possibility of finding the globally optimal circuit is enlarged. As we efficiently compress all possible result circuits as a form of program grammar, we can explore an expansive space within a practical time budget.

Although equality saturation is a time-consuming method in general, we can successfully introduce it for our term rewriting system since most of the homomorphic evaluation circuits have a relatively small scale.

5 EVALUATION

We implemented our method as a tool named Lobster[4]. This section evaluates our Lobster system to answer the following 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 reusing pre-learned rewrite rules?
Q4: What is the benefit of using equality saturation technique?
Q5: What is the benefit of the rule normalization and equational matching?
Q6: How long does Lobster take to obtain optimized circuits?
Q7: How sensitive is Lobster to changes in the training set for learning rewrite rules?

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 RUST, and consists of about 3K lines of code.

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

The offline rule learner collects rewrite rules for online optimization. For each of the benchmarks, we performed the offline learning algorithm (Algo. 1) with a timeout of 1 week before online optimization. We use EUSolver [3][5] and DUET [40][6], which are state-of-the-art open-source search-based synthesizers, for the offline learning task. 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 with a timeout of 12 hour. 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 saturation-based rewriting procedure on the resulting circuit. We used EGG [61] library to implement saturation-based rewriting system. We used learned rewrite rules and commutativity as equality rules, and did not use any eclass analysis.

Circuits optimized by the online optimizer are evaluated by our homomorphic circuit evaluator built using HElib [34].[7] 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

Our benchmarks comprise 25 FHE applications written using the Cingulata APIs shown in Table 1. We had initially collected 64 benchmarks from the following four sources.

- Cingulata benchmarks – 9 FHE-friendly algorithms from diverse domains (medical diagnosis, stream cipher, search, sort) [21].
- Sorting benchmarks – 4 privacy-preserving sorting algorithms (merge, insertion, bubble, and odd-even) [17]. All the sorting algorithms can take up to 6 encrypted 8-bit integers as input.
- Hacker's Delight benchmarks – 26 homomorphic bitwise operations adapted from Hacker's Delight [60][8], 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.

Among these 64 candidate benchmarks, we ruled out 39 benchmarks that are out of reach for homomorphic evaluation even with the state-of-the-art FHE scheme [34], or which are likely depth-optimal based on empirical evidence.

Among the excluded 39 benchmarks, 18 benchmarks have the number of AND/XORs greater than 10,000, or the multiplicative depth is larger than 100. Our homomorphic circuit evaluator runs out of memory for these circuits. The remaining 21 benchmarks are such that (i) baseline optimizer [14] fails to reduce the multiplicative depth, and (ii) we could not mine any rules from their circuit representations even after 7 days of running the offline learner, because this means that even the state-of-the-art synthesizer with practically unlimited time cannot find any improvement.

[5]In our previous work [39], we chose EUSolver among the general-purpose synthesizers that participated in the 2019 SyGuS competition [53] since the tool performs best for our optimization tasks.
[6]We could learn new rewrite rules using DUET, since it outperforms EUSolver for our optimization tasks in most cases.
[7]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.
[8]22 benchmarks used for program synthesis [37] + 4 excerpted from Hacker's Delight [60]

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 [37]	6	87	118
hd-02	absolute value [37]	6	76	229
hd-03	floor of average of two integers (a clever impl.) [37]	5	27	64
hd-04	floor of average of two integers (a naive impl.) [60]	10	75	159
hd-05	max of two integers [37]	7	121	295
hd-06	min of two integers [37]	7	121	295
hd-07	turn off the rightmost contiguous string of 1-bits [37]	5	17	32
hd-08	determine if an integer is a power of 2 [37]	6	18	37
hd-09	round up to the next highest power of 2 [37]	14	134	236
hd-10	find first 0-byte [60]	6	35	73
hd-11	the longest length of contiguous string of 1-bits [60]	18	391	652
hd-12	number of leading 0-bits [60]	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

Baseline

We compare Lobster to 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: $(x \wedge y) \wedge z \longrightarrow x \wedge (y \wedge z)$. In a given circuit c, a substitution σ such that $\sigma((x \wedge y) \wedge z)$ is syntactically matched with a sub-circuit 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: $(x \oplus y) \wedge z \longrightarrow (x \wedge z) \oplus (y \wedge z)$. 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 termination condition is satisfied. As the tool is not publicly available, we reimplemented their algorithm.[9]

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

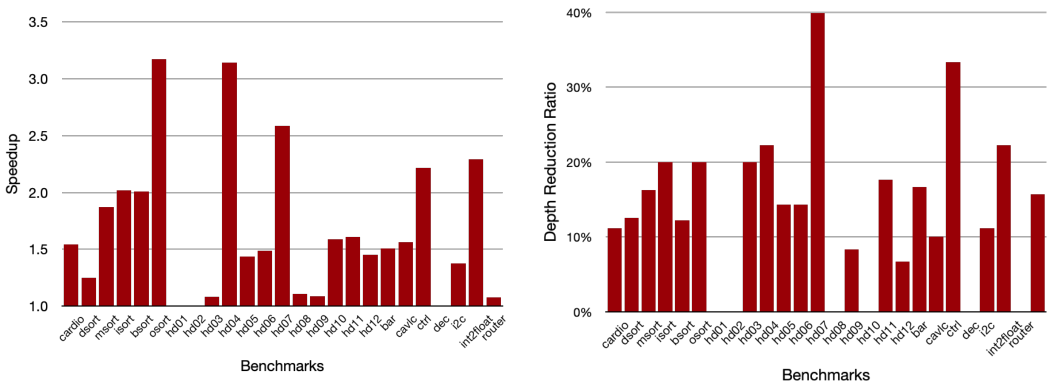

Fig. 6. Main results comparing the optimization performance of Lobster and Carpov et al. [14] – Speedups in overall homomorphic evaluation time (left) and depth reduction ratios (right).

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 24 benchmarks. Both of the tools are given the timeout limit of 12 hours for the online optimization tasks; in case of exceeding the limit, we use the best intermediate results computed so far.

We measure Lobster's reduction ratios of the multiplicative depth and speedups in overall homomorphic evaluation time against the initial Cingulata-generated circuits, and compared them with Carpov et al. [14]. The results can be found in Table 2. Lobster is able to optimize 22 out of 25 benchmarks within the timeout limit. Lobster achieves 1.08x – 5.43x speedups with the geometric mean of 2.05x. The number of AND gates increases up to 1.9x more with the geometric mean of 1.31x. The depth reduction ratios range from 12.5% to 53.3% with the geometric mean of 25.1%. Fig. 6 illustrates the summarized results comparing the optimization performance of Lobster and Carpov et al. [14].

We next study the results in detail. Most notably, Lobster achieves 2.62x and 1.60x 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. For the four sorting benchmarks, we used single-path term rewriting, since EGG library failed to perform saturation task for circuits that has multiplicative depth over 25. Lobster reduces the depth by 20% for each of them. The osort benchmark shows a 3.17x speedup, and the other three benchmarks show 2.0x speedups. As of the Hacker's Delight benchmarks, 10 out of 12 observe improvements. The speedups for hd-04, hd-07, hd-08 and hd-10 are remarkable (3.8x, 2.6x, 2.4x and 3.3x, respectively). For the other benchmarks, we observe 1.08x – 1.89x speedups. However, both of the two optimizers fail to optimize the other 2 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 most of these benchmarks to be depth-optimal.

Table 2. Detailed main results. 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 depth and evaluation time is the same as the original).

	Original Cingulata		**Carpov et al. [14]**			**Lobster**		
Name	**×Depth**	**Eval. Time**	**×Depth**	**#AND ↑**	**Eval. Time**	**×Depth**	**#AND ↑**	**Eval. Time**
cardio	10	17m 14s	9	**x1.07**	10m 08s	**8**	x1.12	**6m 34s**
dsort	9	10m 52s	8	**x1.08**	8m 29s	**7**	x1.33	**6m 47s**
msort	45	5h 20m 59s	41	**x1.02**	5h 00m 06s	**36**	x1.88	**2h 40m 23s**
isort	45	5h 20m 16s	-	-	-	**36**	**x1.88**	**2h 38m 53s**
bsort	45	5h 21m 46s	41	**x1.02**	5h 06m 09s	**36**	x1.88	**2h 32m 38s**
osort	25	2h 16m 58s	-	-	-	**20**	**x1.91**	**43m 12s**
hd01	6	4m 36s	-	-	-	-	-	-
hd02	6	4m 50s	-	-	-	-	-	-
hd03	5	1m 08s	-	-	-	**4**	**x1.44**	**1m 03s**
hd04	10	9m 06s	9	**x1.00**	7m 36s	**7**	x1.31	**2m 25s**
hd05	7	6m 08s	-	-	-	**6**	**x1.52**	**4m 16s**
hd06	7	6m 14s	-	-	-	**6**	**x1.54**	**4m 12s**
hd07	5	1m 02s	-	-	-	**3**	**x1.12**	**24s**
hd08	6	2m 18s	5	x1.00	1m 03s	**5**	**x1.00**	**57s**
hd09	14	13m 03s	12	**x1.10**	9m 34s	**11**	x1.37	**8m 48s**
hd10	6	4m 24s	5	x1.03	2m 07s	**5**	**x1.00**	**1m 20s**
hd11	18	33m 31s	17	**x1.00**	28m 30s	**14**	x1.08	**17m 42s**
hd12	16	22m 31s	15	**x1.00**	18m 01s	**14**	x1.12	**12m 26s**
bar	12	56m 55s	-	-	-	**10**	**x0.89**	**37m 47s**
cavlc	16	26m 35s	10	x1.20	15m 01s	**9**	**x1.18**	**9m 37s**
ctrl	8	3m 06s	6	**x1.02**	2m 44s	**4**	x1.19	**1m 14s**
dec	3	38s	-	-	-	-	-	-
i2c	15	51m 00s	9	**x1.08**	21m 38s	**8**	x1.21	**15m 45s**
int2float	15	15m 23s	9	**x1.13**	6m 30s	**7**	x1.21	**2m 50s**
router	19	37m 26s	10	**x1.31**	12m 34s	**10**	x1.38	**11m 39s**

As of the EPFL benchmarks, 6 out of 7 observe improvements. Both optimizers fail to optimize dec, which is relatively simple. For bar, we observe 1.51x speedup. For the other benchmarks (cavlc, ctrl, i2c, int2float and router), Lobster achieves remarkable speedups (2.5x – 5.4x).

The number of AND gates increases 1.31x more on average. For the 4 sorting benchmarks ({m,i,b,o}sort), we observe nearly 2x increases. For the other benchmarks, we observe up to 1.5x increases. These increases are acceptable considering depth reduction ratio and speedup. 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 better than Carpov et al. [14] within given time limit (12 hours).

Note that Fig. 7 shows 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). This shows that multiplicative depth reduction is a good proxy for speedup, and thus we only measured depth reduction ratio rather than speedup in sub-experiments (Section 5.4 – Section 5.8).

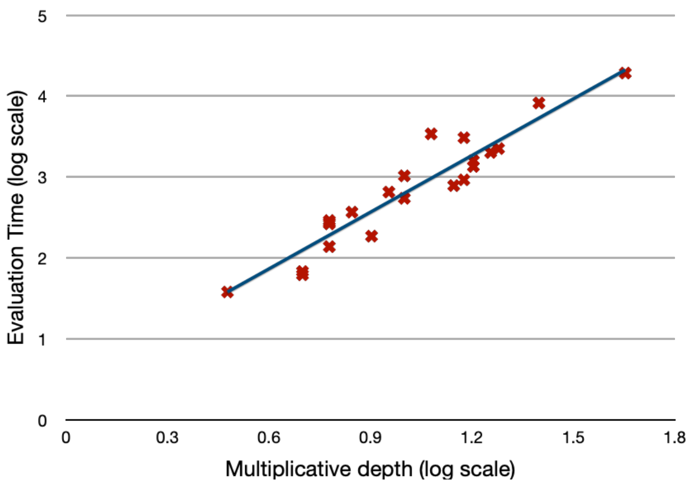

Fig. 7. Correlation log-log plot of multiplicative depth and homomorphic evaluation time

Learning Capability

We investigate the learned rewrite rules. From all the benchmarks, our rule learner mines 502 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 14 and 13, respectively. Fig. 8 shows how often these rules were applied to reduce the multiplicative depth during our single-path term rewriting. Relatively small-sized rules (size $5-15$) are most frequently used, but also the large-sized rules are sometimes applied and optimize wide areas of the input boolean circuits.

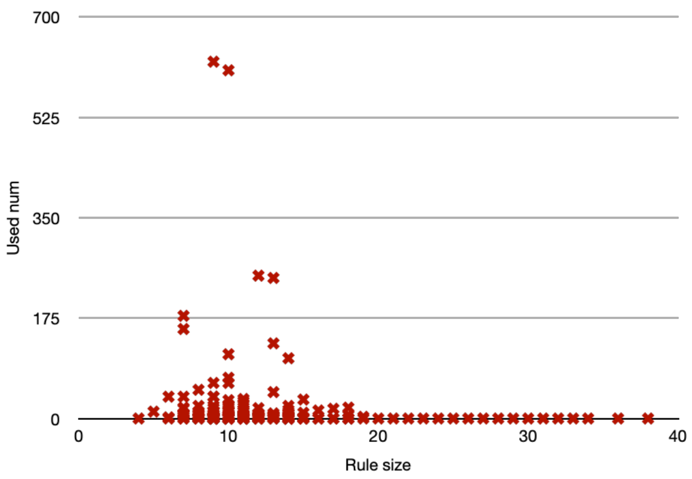

Fig. 8. Distribution of rule sizes and how often they were used during optimization

The machine-found optimization patterns are surprisingly aggressive. For example, the following intricate rules enable to reduce the depth of a rewritten path by 1 when applied once (we denote $1 \oplus c$ as $\neg c$).

$$(v_1 \land (v_2 \land ((v_3 \oplus (v_4 \land v_5)) \oplus (v_6 \land v_5))))$$
$$\rightarrow ((((v_6 \oplus v_4) \land v_5) \oplus v_3) \land (v_2 \land v_1))$$

$$((\neg((v_1 \land (\neg(v_2 \oplus v_3))) \oplus (v_2 \oplus v_3))) \land v_4)$$
$$\rightarrow (((\neg v_2) \oplus v_3) \land ((\neg v_1) \land v_4))$$

$$(\neg((\neg((((v_1 \oplus v_2) \land v_3) \land v_4) \land v_5)) \oplus v_2))$$
$$\rightarrow ((((v_2 \oplus v_1) \land v_4) \land (v_3 \land v_5)) \oplus v_2)$$

$$((\neg((v_1 \oplus (v_2 \land v_3)) \oplus (v_4 \land v_3))) \land v_5)$$
$$\rightarrow (((v_2 \oplus v_4) \land (v_5 \land v_3)) \oplus ((\neg v_1) \land v_5))$$

$$((((v_1 \oplus v_2) \oplus v_3) \land (((v_1 \oplus v_2) \land v_3) \oplus (v_1 \land v_2)))\land$$
$$(((((v_1 \oplus v_2) \land v_3) \oplus (v_1 \land v_2)) \land ((v_1 \oplus v_2) \land v_3))\oplus$$
$$(\neg((v_1 \oplus v_2) \land v_3))))$$
$$\rightarrow ((v_3 \land v_1) \land v_2)$$

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).

Answer to Q1: Lobster can optimize 22 out of 25 realistic FHE applications. (x2.26 speedup, 25.1% depth reduction).

5.3 Comparison to the Baseline

Fig. 6 shows that Lobster significantly outperforms the existing state-of-the-art homomorphic circuit optimizer [14] in terms of both depth reduction ratio and homomorphic evaluation time. Only 15 out of 25 benchmarks can be optimized by Carpov et al. [14], whereas Lobster is able to optimize 22. Compared to Carpov et al. [14], Lobster's speedup is increased by up to 3.17x with the geometric mean of 1.56x. The depth reduction ratio is increased by up to 40.0% with the geometric mean of 13.8%.

We observe that Carpov et al. [14] needs relatively small amount of optimization time than Lobster. It took 1 second – 25 minutes to optimize benchmarks with the average of 2 minutes, whereas Lobster took 12 hours to optimize each benchmark. This is because Carpov et al. [14] uses single-path rewriting with two simple rewrite rules, whereas Lobster uses saturation-based rewriting with total 502 rewrite rules.

We empirically observe that Carpov et al. [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.

Answer to Q2: Lobster outperforms existing FHE optimizer. (x1.56 speedup, 13.8% depth reduction ratio)

5.4 Efficacy of Reusing Pre-Learned Rewrite Rules

We observe that reusing pre-learned rewrite rules significantly enhanced Lobster's scalability and exploration power.

To investigate the benefit for scalability, we compare Lobster to a simple method that uses the offline rule learner as an on-the-fly optimizing synthesizer. Since it does not use any pre-learned rewrite rules, it can not use saturation-based rewriting. While performing single-path rewriting, it finds an optimized version of sub-circuits using a program synthesizer rather than matching it with pre-learned rewrite rules. The timeout limit for optimization is again set to 12 hours, and we use the best intermediate results when the budget expires.

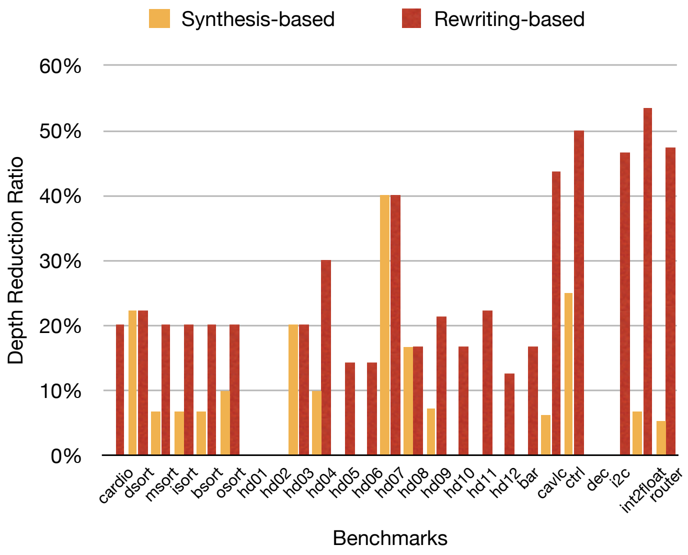

Fig. 9. Comparison between on-the-fly synthesis and equality saturation with learned rules

Fig. 9 summarizes the results. The synthesis-based optimizer can optimize only 14 benchmarks within the timeout limit. Furthermore, in all the 14 benchmarks, the depth reduction ratio is less than that of LOBSTER that reuses pre-learned rewrite rules (geometric mean of 8.2% vs 25.1%). That is mainly 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.

To investigate the benefit for exploration power, we compare LOBSTER to a simple version that only uses boolean ring theory (Example. 4.3) as rewrite rules. Fig. 10 summarizes the results. The simple version of LOBSTER can optimize only 6 benchmarks within the timeout limit. Furthermore, in all the 6 benchmarks, the depth reduction ratio is less than that of LOBSTER that uses pre-learned aggressive rewrite rules. This shows that LOBSTER can efficiently escape local optima by applying pre-learned rewrite rules even though they can be induced by boolean ring theory.

We also investigate that adding new rewrite rules can enhance exploration power. We compare LOBSTER to a simple version that only uses 188 rewrite rules learned by EUSOLVER rather than the whole 502 rewrite rules learned by EUSOLVER and DUET. Fig. 10 summarizes the results. In five benchmarks, the simple version's depth reduction ratio is less than that of LOBSTER that uses all rewrite rules. In the exceptional case of hd 01, the simple version can reduce multiplicative depth by 1, whereas LOBSTER can not optimize it within time limit. This is because full version of LOBSTER needs much more time to perform each iteration step for equality saturation, since it uses nearly 3 times more rewrite rules than the simple version. If the LOBSTER is given sufficient amount of time limit, it can also optimize hd 01.

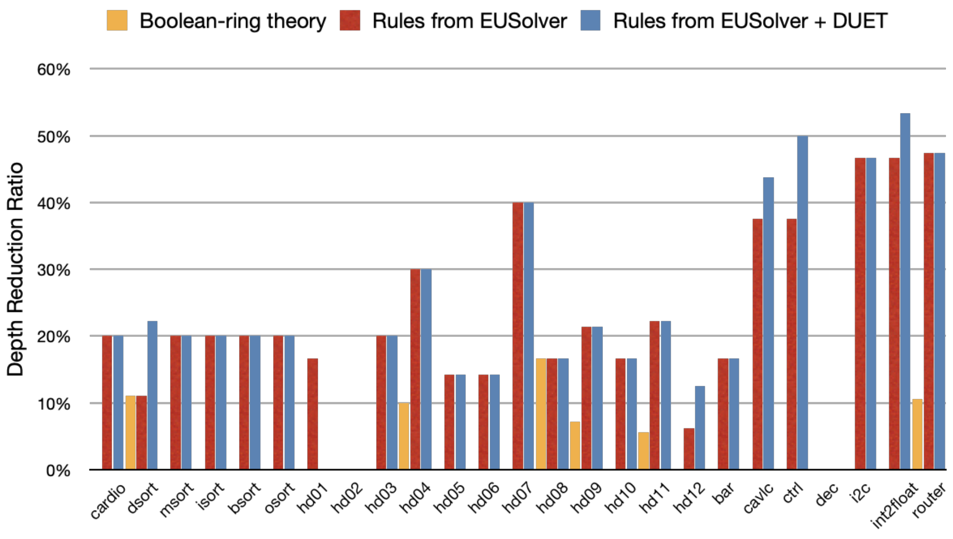

Fig. 10. Impact of changing rewrite rules

Answer to Q3: Reusing learned rules enhances LOBSTER's scalability and exploration power. (1 week vs 12 hour opt. time, 2.6% vs 23.7% vs 25.1% depth reduction)

5.5 Efficacy of Equality Saturation

We now evaluate the effectiveness of saturation-based rewriting, which we used for online optimization. We compare LOBSTER to a previous version [39] that uses single-path rewriting only. Both of the tools use the same pre-learned rewrite rules.

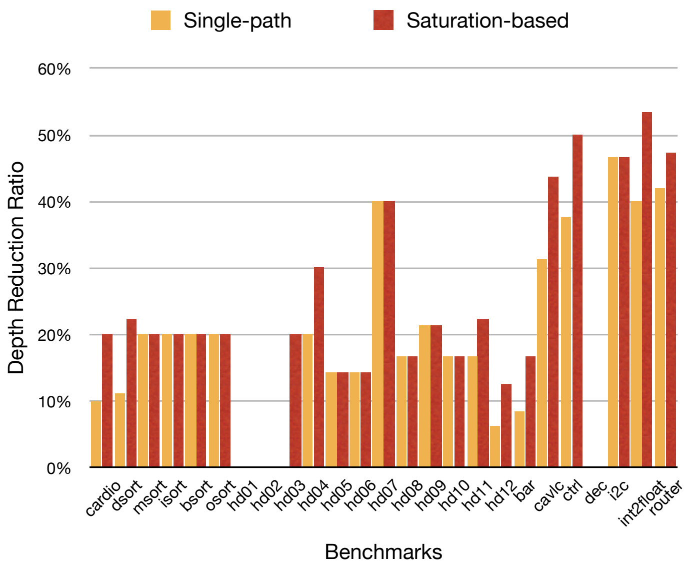

Fig. 11. Efficacy of Equality Saturation

Fig. 11 summarizes the results. Although both of the tools can optimize 22 benchmarks within time limit, previous version's depth reduction ratio is less than that of saturation-based rewriting version in 11 benchmarks. This shows that saturation-based rewriting can explore wider area of optimization results because it can store every possible rewriting sequences (i.e. saturation-based rewriting obtains backtracking effect).

Answer to Q4: Equality saturation enhances Lobster's exploration power via backtracking. (20.2% vs 25.1% depth reduction)

5.6 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 matching instead of equational matching when conducting term rewriting and applies the learned rules without the normalization process.

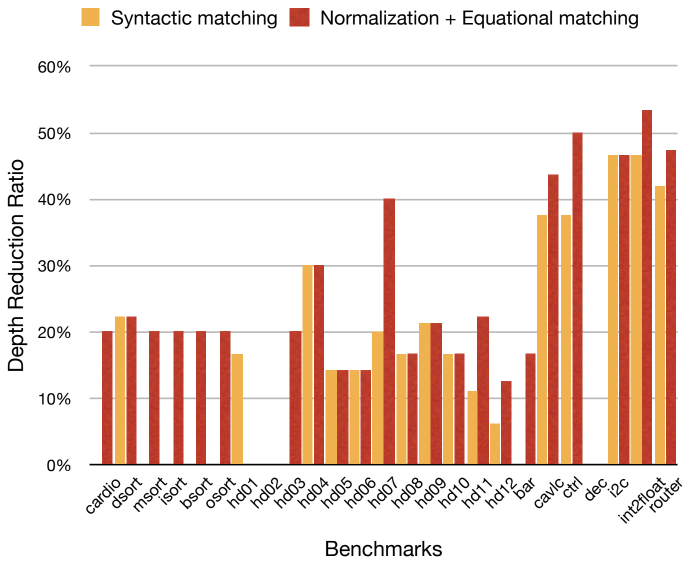

Fig. 12. Efficacy of equational rewriting

Fig. 12 summarizes the results. The variant can optimize 16 benchmarks (Lobster can optimize 22), and its depth reduction ratio is less than that of Lobster in 12 benchmarks. In the exceptional case of hd 01, the variant can reduce multiplicative depth by 1, whereas Lobster can not optimize it within time limit. This is because of the difference of time cost for each iteration step for equality saturation. Same as 5.4, Lobster can also optimize hd 01 if it is given sufficient amount of time limit. We conclude that overall, the rule normalization and equational term rewriting play crucial roles in giving flexibility to the rewriting procedure.

Answer to Q5: Equational matching enables flexible rewriting and enhances exploration power. (16 vs 22 optimized benchmarks, 17.6% vs 25.1% depth reduction)

5.7 Sensitivity to Changes in a Time Limit

We now investigate the effects of changing the time limit of online optimization. We compared Lobster with its variant that is given 1 hour of time limit.

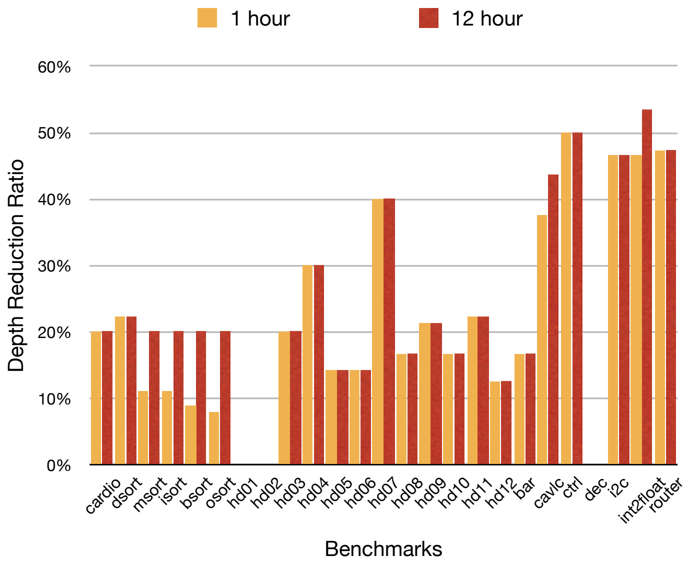

Fig. 13. Comparison between the optimization results with 1h and 12h of time limit.

Fig. 13 summarizes the results. Both of the tools can optimize 22 benchmarks within time limit. In 4 sorting benchmarks ({m,i,b,o}sort) that use single-path rewriting, Lobster significantly outperforms the variant. In the other benchmarks that use saturation-based rewriting, Lobster slightly outperforms the variant. This shows that most of the effective iteration steps for equality saturation are performed within 1 hour, since each iteration step needs much more time as the number of iteration grows. We conclude that the most appropriate time limit for Lobster is 12 hours, but we can also get similar optimization result with 1 hour of time limit in saturation-based rewriting.

Answer to Q6: Lobster takes less than 12 hour to obtain practically saturated circuit.

5.8 Sensitivity to Changes in a Training Set

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. 14 are testing benchmarks, and the other 11 benchmarks are training benchmarks.

As can be seen in Fig. 14 that summarizes the results, the smaller set of training programs does not lead to significant performance degradation. The cardio, hd05, hd06, hd09, hd12, cavlc, and int2float 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 much sensitive to changes in a given set of training programs.

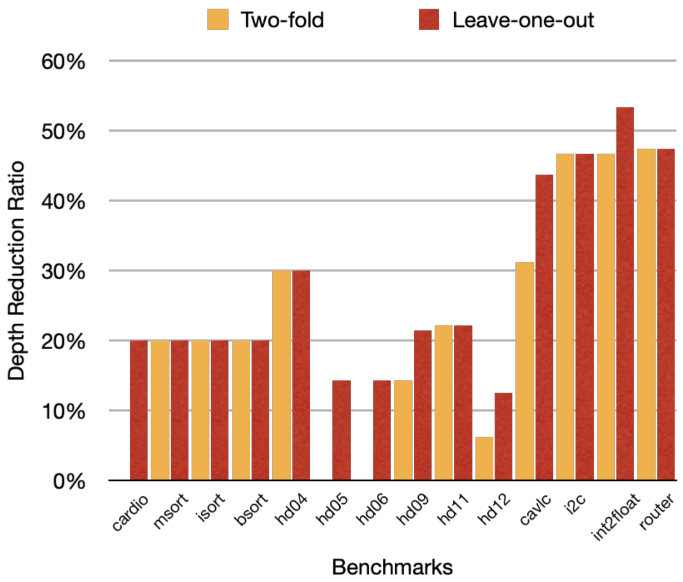

Fig. 14. Comparison between the optimization results with two-fold cross validation and leave-one-out cross validation.

Answer to Q7: Lobster is not critically sensitive to changes in a given set of training circuits. (11 vs 14 optimized benchmarks, 23.6% vs 29.0% depth reduction)

6 RELATED WORK

Existing FHE Compilers in comparison

Existing FHE compilers [4, 15, 22–25, 43] use fixed, hand-tuned optimization methods. These compilers 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 methods are hand-tuned, 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. It supports optimization of circuits for reducing multiplicative depth based on hand-written rules. Cingulata uses ABC [12], an open-source boolean circuit optimizer, but it does not directly address our optimization problem [15]. 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 Carpov et al. [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 Carpov et al. [14] does not perform very well for such a hard optimization task.

- RAMPARTS [4] is an optimizing 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 and hand-written rules. It automatically selects the parameters of FHE schemes and the plain text encoding for input values and uses a number of hand-written circuit optimization rules for reducing multiplicative depth.

- ALCHEMY [23] 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 [25] is a domain-specific optimizing compiler for FHE applications on neural network inference. It enables a number of optimizations automatically, but they are hand-tuned and 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.

- COPSE [43] is a domain-specific optimizing compiler for FHE applications on decision forest inference. It vectorizes decision-forest inference models (i.e. parallelizes operations performed during inference) to exploit ciphertext packing technique and optimize FHE applications. This vectorizing process also minimizes the multiplicative depth growth, but its method is hand-tuned and specific to decision forest inference.

- EVA [24] is an optimizing compiler for arithmetic FHE applications. It enables a hand-tuned optimization specific to the CKKS FHE scheme by lowering the cost overhead caused by crypto operations (e.g. linearize, rescale, relevel) that ensure the safety of homomorphic evaluations. By contrast, our optimization framework is scheme-unaware and has a larger potential for speedup since it aims to reduce the multiplicative depth of FHE applications.

- Porcupine [22] is an optimizing compiler for arithmetic FHE applications. Similar to ours, it uses program synthesis to optimize FHE applications. However, it targets specific DSL named Quill rather than boolean circuits and user has to provide hand-written sketch to successfully optimize target applications.

Superoptimization

Similar to ours, existing superoptimizers [7, 13, 35, 49, 50] 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 [49] is similar to Bansal and

Aiken [7] but is based on a constraint-based synthesis technique and targets a subset of LLVM IR. STOKE [35, 50] 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 [28–31, 52, 58, 62]. 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. [28], 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 and Equality Saturation

Term rewriting [9, 11, 20, 55, 57] and equality saturation [45, 54, 59, 61, 63] has been widely used in program transformation systems. 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. [57] used term rewriting in ML compilers and presented a language for writing rewriting strategies. Tate et al. [54] and Yang et al. [63] used equality saturation(i.e. saturation-based term rewriting) with hand-tuned rewrite rules to optimize C-like languages and trained DNN models respectively. 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.

Similar to ours, Ruler [46] used equality saturation to automatically infer rewrite rules for a given user-defined domain. Although Ruler found 35 rewrite rules for the boolean circuit domain, we observed that the saturation-based term rewriting with these 35 rewrite rules shows little optimization effect for homomorphic evaluation. It just slightly outperforms an ablation of Lobster used in Section 5.4 that uses Boolean ring theory as rewrite rules (2.6% vs 3.3%). This is because rewrite rules inferred by Ruler have no objective in mind to reduce the multiplicative depth.

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 equational term rewriting on the new, unseen circuit based on the equality saturation to maximally leveraging the learned rules. We demonstrated the effectiveness of our method with 25 FHE applications from diverse domains. The results show that our method achieves sizeable optimizations that are complementary to existing domain-specific optimization techniques.

Though we target a specific kind of optimization tasks for homomorphic evaluation in this paper, we believe our approach is potentially applicable to other optimization tasks. Our method of synthesizing optimization rules and exhaustively applying the combinations of the learned optimization rules in a cost-effective way by the time-bounded equality saturation technique can be beneficial to a broader class of optimization tasks.

ACKNOWLEDGMENTS

Woosuk Lee and Kwangkeun Yi are corresponding authors. This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337, (SW STAR LAB) Research on Highly-Practical Automated Software Repair & No. 2021-0-00758 & No. 2022-0-00995), National Research Foundation of Korea (No. 2020R1C1C1014518 & No. 2021R1A5A1021944), BK21 FOUR Intelligence Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea(NRF) (4199990214639), Samsung Electronics Co.,Ltd (IO220411-09496-01), and Greenlabs (No. 0536-20220078).

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 (London, United Kingdom) (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 (San Jose, California, USA) (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 (Cambridge, Massachusetts) (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 (Edinburgh, UK) (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 (Singapore, Republic of Singapore) (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 (Lisbon, Portugal) (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] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T Lee, and Brandon Reagen. 2021. Porcupine: a synthesizing compiler for vectorized homomorphic encryption. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 375–389.

[23] 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 (Toronto, Canada) (CCS '18). ACM, New York, NY, USA, 1020–1037. https://doi.org/10.1145/3243734.3243828

[24] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM. https://doi.org/10.1145/3385412.3386023

[25] 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 (Phoenix, AZ, USA) (PLDI '19). ACM, New York, NY, USA, 142–156. https://doi.org/10.1145/3314221.3314628

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

[27] 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 (New York, NY, USA) (ICML '16). JMLR.org, 201–210. http://dl.acm.org/citation.cfm?id=3045390.3045413

[28] 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.

[29] 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 (Barcelona, Spain) (PLDI '17). ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[30] 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 (Paris, France) (POPL '17). ACM, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

[31] 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 (Portland, OR, USA) (PLDI '15). ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

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

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

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

[35] 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 (Santa Barbara, CA, USA) (PLDI '16). ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

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

[37] 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 (Cape Town, South Africa) (ICSE '10). ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[38] 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

[39] 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 Conference on Programming Language Design and Implementation (London, UK) (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 503–518. https://doi.org/10.1145/3385412.3385996

[40] Woosuk Lee. 2021. Combining the Top-down Propagation and Bottom-up Enumeration for Inductive Program Synthesis. Proc. ACM Program. Lang. 5, POPL, Article 54 (Jan. 2021), 28 pages. https://doi.org/10.1145/3434335

[41] 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.

[42] 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.

[43] Raghav Malik, Vidush Singhal, Benjamin Gottfried, and Milind Kulkarni. 2021. Vectorized Secure Evaluation of Decision Forests. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) (PLDI 2021). Association for Computing Machinery, New York, NY, USA, 1049–1063. https://doi.org/10.1145/3453483.3454094

[44] 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 (Chicago, Illinois, USA) (CCSW '11). ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2046660.2046682

[45] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (Jun 2020). https://doi.org/10.1145/3385412.3386012

[46] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. Proc. ACM Program. Lang. 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

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

[48] 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.

[49] 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

[50] 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 (Houston, Texas, USA) (ASPLOS '13). ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

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

[52] 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 (Seattle, Washington, USA) (PLDI '13). ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

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

[54] 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 (Savannah, GA, USA) (POPL '09). ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[55] 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

[56] 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 (Toronto, Canada) (WAHC '18). ACM, New York, NY, USA, 49–60. https://doi.org/10.1145/3267973.3267978
[57] 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 (Baltimore, Maryland, USA) (ICFP '98). ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/289423.289425
[58] 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 (Barcelona, Spain) (PLDI '17). ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365
[59] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. arXiv:2002.07951 [cs.DB]
[60] Henry S. Warren. 2012. Hacker's Delight (2nd ed.). Addison-Wesley Professional.
[61] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. Proceedings of the ACM on Programming Languages 5, POPL (Jan 2021), 1–29. https://doi.org/10.1145/3434304
[62] 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
[63] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. arXiv:2101.01332 [cs.AI]

Proofs Before describing the proofs of Theorem 1 and 2, we begin with preliminary concepts from Baader and Nipkow [6].

Definition 7.1. Let $\equiv$ be a binary relation on $T_{\Sigma,X}$.

(1) The relation $\equiv$ is closed under substitutions iff $s \equiv t$ implies $\sigma(s) \equiv \sigma(t)$ for all $s, t \in T_{\Sigma,X}$ and substitutions σ.
(2) The relation $\equiv$ is compatible with Σ-context iff $s \equiv s'$ implies $t[p \leftarrow s] \equiv t[p \leftarrow s']$ for all $t \in T_{\Sigma,X}$ and positions $p \in Pos(t)$.

Lemma 7.2. *$\approx_{\mathcal{R}}$ is closed under substitutions and compatible with Σ-context.*

Proof. By Theorem 3.1.12 in Baader and Nipkow [6], for any set E of Σ-identities, the relation $\leftrightarrow_E^*$ is closed under substitutions. By Lemma 3.1.11 in Baader and Nipkow [6], the relation $\leftrightarrow_E^*$ is also compatible with Σ-context. Setting $E = \mathcal{R}$ finishes the proof. □

Lemma 7.3. *For all $s, t \in T_{\Sigma,X}$,*

$$s \approx_C t \implies s \approx_{\mathcal{R}} t.$$

Proof. Straightforward from the fact that $C \subseteq \mathcal{R}$. □

Now we are ready to prove Theorem 1.

Theorem 1

$$\forall s, t \in T_{\Sigma,X}.\ s \rightarrow_E t \Rightarrow s \approx_{\mathcal{R}} t$$

(Stated in Section 4.3)

Proof. By the definition of $s \rightarrow_E t$ and E, there exist $(l \approx_{\mathcal{R}} r) \in E, p \in Pos(s), \sigma.$ such that $s\ |_p \approx_C \sigma(l), t = s[p \leftarrow \sigma(r)]$.

$$\begin{array}{lr}
s\ |_p \approx_{\mathcal{R}} \sigma(l) & \text{(By Lemma 7.3 and } s\ |_p \approx_C \sigma(l)) \\
s[p \leftarrow s\ |_p] \approx_{\mathcal{R}} s[p \leftarrow \sigma(l)] & \text{(By Lemma 7.2)} \\
\sigma(l) \approx_{\mathcal{R}} \sigma(r) & \text{(By Lemma 7.2 and } l \approx_{\mathcal{R}} r) \\
s[p \leftarrow \sigma(l)] \approx_{\mathcal{R}} s[p \leftarrow \sigma(r)] & \text{(By Lemma 7.2)}
\end{array}$$

Therefore, $s \approx_{\mathcal{R}} t$. □

Next, before describing the proof of termination, we define the following strict orders between terms.

Definition 7.4. For all $s, t \in T_{\Sigma,X}$,

$$s \succ t \iff (\ell(s) > \ell(t)) \lor (\ell(s) = \ell(t) \land |\mathcal{CP}(s)| > |\mathcal{CP}(t)|).$$

Recall that by replacing an old circuit region compassing a critical path with a new circuit region of smaller depth, we can either i) decrease the overall depth if the critical path is unique, or ii) decrease the number of parallel critical paths, which is reflected on the definition of $\succ$.

The following lemma tells us that the order $\succ$ is compatible with Σ-context.

Lemma 7.5. *Suppose we have $s, t_1, t_2 \in T_{\Sigma,X}$ such that $s\ |_p = t_1$ and $\ell(t_1) > \ell(t_2)$. Then,*

$$\forall p \in \mathcal{CP}(s).\ s[p \leftarrow t_1] \succ s[p \leftarrow t_2].$$

Proof.
- Case $p = \epsilon$: $s[p \leftarrow t_1] = t_1 \succ t_2 = s[p \leftarrow t_2]$.
- Case $p = p'.1$:
  Suppose $s\ |_{p'} = \land(t_1, s_2)$ which makes $s\ |_p = t_1$. By the fact $p = p'.1$ and the definition of $\mathcal{CP}$, $\ell(t_1) \geq \ell(s_2)$. There are two cases.
  Case (1) $\ell(t_1) > \ell(s_2)$:

$$\begin{aligned}
&\ell(\land(t_1, s_2)) = 1 + \ell(t_1) && \\
&\ell(\land(t_2, s_2)) = 1 + \max(\ell(t_2), \ell(s_2)) && \text{(By Def. of } \ell) \\
&\ell(t_1) > \ell(t_2) && \text{(By the premise)} \\
&\ell(t_1) > \ell(s_2) && \text{(By the case assumption)} \\
&\therefore \ell(s[p \leftarrow t_1]) > \ell(s[p \leftarrow t_2]) &&
\end{aligned}$$

  Case (2) $\ell(t_1) = \ell(s_2)$:

$$\begin{aligned}
&\ell(\land(t_1, s_2)) = 1 + \ell(s_2) && \\
&\ell(\land(t_2, s_2)) = 1 + \ell(s_2) && (\ell(s_2) = \ell(t_1) > \ell(t_2)) \\
&|\mathcal{CP}(\land(t_1, s_2))| = |\mathcal{CP}(t_1) \uplus \mathcal{CP}(s_2)| \times 2 && \text{(By Def. of } \mathcal{CP}) \\
&|\mathcal{CP}(\land(t_2, s_2))| = |\mathcal{CP}(s_2)| \times 2 && \\
&\mathcal{CP}(t_1) \neq \emptyset && \\
&|\mathcal{CP}(\land(t_1, s_2))| > |\mathcal{CP}(\land(t_2, s_2))| &&
\end{aligned}$$

  Note that $\mathcal{CP}(t_1) \neq \emptyset$ because otherwise, $\ell(t_1) = 0 > \ell(t_2) \geq 0$ which leads to a contradiction. Therefore, $\ell(s[p \leftarrow t_1]) = \ell(s[p \leftarrow t_2]) \land |\mathcal{CP}(s[p \leftarrow t_1])| > |\mathcal{CP}(s[p \leftarrow t_2])|$.
  The other case where $s\ |_{p'} = \oplus(t_1, s_2)$ can be proven similarly.
- Case $p = p'.2$: Similar to the above case.

□

Now we are ready to prove Theorem 2.

Theorem 2$\rightarrow_E$ is a terminating relation.
(Stated in Section 4.3)

Proof. Straightforward from Lemma 7.5 and the fact that $\succ$ is a strict order. □

Matching Algorithm

$$\frac{\{s \doteq^? s\} \cup P; S}{P; S} \qquad \text{[Trivial]}$$

$$\frac{\{f(s_1, s_2) \doteq^? f(t_1, t_2)\} \cup P; S}{\{s_1 \doteq^? t_1, s_2 \doteq^? t_2\} \cup P; S} \qquad \text{[Decomposition]}$$

$$\frac{\{f(s_1, s_2) \doteq^? f(t_1, t_2)\} \cup P; S}{\{s_2 \doteq^? t_1, s_1 \doteq^? t_2\} \cup P; S} \qquad \text{[C-Decomposition]}$$

$$\frac{\{f(s_1, s_2) \doteq^? g(t_1, t_2)\} \cup P; S}{\bot}\ f \neq g \qquad \text{[Symbol Clash]}$$

$$\frac{\{f(s_1, s_2) \doteq^? x\} \cup P; S}{\bot} \qquad \text{[Symbol-Variable Clash]}$$

$$\frac{\{x \doteq^? t_1\} \cup P; \{x \doteq t_2\} \cup S}{\bot}\ t_1 \neq t_2 \qquad \text{[Merging Clash]}$$

$$\frac{\{x \doteq^? t\} \cup P; S}{P; \{x \doteq t\} \cup S}\ x \doteq t' \notin S \text{ where } t \neq t' \qquad \text{[Variable Elimination]}$$

Table 3. Rules for C-matching

E-Matching Algorithm. Solving a matching problem for two terms s and t is represented by $S = \{s \approx_E^? t\}$. A conventional E-matching algorithm derives a set of equations in solved form:

$$\{x_1 \approx_E t_1, \cdots, x_n \approx_E t_n\}$$

where all x_i's are pairwise distinct.

Matching System. The symbol $\bot$ or a pair $P; S$ where

- P is a set of matching problems,
- S is a set of equations in matched form.
- $\bot$ represents failure (i.e., no matchers).

A matcher (or a solution) of a system $P; S$ returns a matcher that solves each of the matching equations in P and S.

Table. 3 depicts an example of the matching rules when $E = C$.

The following algorithm matchs s to t.

(1) Create an initial system is $\{s \doteq^? t\}; \emptyset$.
(2) Apply successively the matching rules.
(3) If the final system is $\emptyset; S$, return S.
(4) If the final system is $\bot$, then fail.

C-Matching Algorithm.

Example 7.6. The followings show the process of finding C-matchers of $f(x, f(a, x))$ and $f(g(a), f(a, g(a)))$ where f and g are function symbols, and the others are variables.

$$\begin{gathered}
\{f(x, y) \doteq_C^? f(f(a, b), f(b, a))\}; \emptyset \Longrightarrow_{\text{Decomposition}} \\
\{x \doteq_C^? f(a, b); y \doteq_C^? f(b, a)\}; \emptyset \Longrightarrow_{\text{V.Elim}} \\
\{y \doteq_C^? f(b, a)\}; \{x \doteq_C f(a, b)\} \Longrightarrow_{\text{V.Elim}} \\
\emptyset; \{x \doteq_C f(a, b), y \doteq_C f(b, a)\} \Longrightarrow_{\text{V.Elim}}
\end{gathered}$$

The other way is

$$\begin{aligned}
\{f(x,y) \doteq^?_C f(f(a,b), f(b,a))\}; \emptyset &\Longrightarrow_{\mathsf{C-Decomposition}} \\
\{x \doteq^?_C f(b,a); y \doteq^?_C f(a,b)\}; \emptyset &\Longrightarrow_{\mathsf{V.Elim}} \\
\{y \doteq^?_C f(a,b)\}; \{x \doteq_C f(b,a)\} &\Longrightarrow_{\mathsf{V.Elim}} \\
\emptyset; \{x \doteq_C f(b,a), y \doteq_C f(a,b)\} &\Longrightarrow_{\mathsf{V.Elim}}
\end{aligned}$$

Note that C-Decomposition and Decomposition transform the same system in different ways. There may exist multiple matchers, and C-matching algorithm is NP-complete [38].

To avoid the exponential complexity of C-matching algorithm, we bound the number of applications of the C-Decomposition rule. This lead to incompleteness, but does not harm the correctness of the matching algorithm.