Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting

Jaehyung Lee
Department of Computer Science & Engineering
Hanyang University
Ansan, Korea
huna3869@hanyang.ac.kr

Woosuk Lee*
Department of Computer Science & Engineering
Hanyang University
Ansan, Korea
woosuk@hanyang.ac.kr

ABSTRACT

Mixed Boolean Arithmetic (MBA) obfuscation transforms a program expression into an equivalent but complex expression that is hard to understand. MBA obfuscation has been popular to protect programs from reverse engineering thanks to its simplicity and effectiveness. However, it is also used for evading malware detection, necessitating the development of effective MBA deobfuscation techniques. Existing deobfuscation methods suffer from either of the four limitations: (1) lack of general applicability, (2) lack of flexibility, (3) lack of scalability, and (4) lack of correctness. In this paper, we propose a versatile MBA deobfuscation method that synergistically combines program synthesis, term rewriting, and an algebraic simplification method. The key novelty of our approach is that we perform *on-the-fly learning* of transformation rules for deobfuscation, and apply them to rewrite the input MBA expression. We implement our method in a tool called ProMBA and evaluate it on over 4000 MBA expressions obfuscated by the state-of-the-art obfuscation tools. Experimental results show that our method outperforms the state-of-the-art MBA deobfuscation tool by a large margin, successfully simplifying a vast majority of the obfuscated expressions into their original forms.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; • **Theory of computation** → Equational logic and rewriting.

KEYWORDS

Program Synthesis; Mixed Boolean Arithmetic Obfuscation; Term Rewriting

ACM Reference Format:
Jaehyung Lee and Woosuk Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Proceedings of Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623186

*Corresponding author

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623186

1 INTRODUCTION

Code obfuscation aims to safeguard program assets from reverse engineering through the application of semantics-preserving transformations to the source code. A desirable obfuscation technique, from the obfuscator's perspective, should preserve the program's semantics at a reasonable obfuscation cost and runtime overhead, while also making the program resistant to both manual and automated reverse engineering.

Mixed Boolean Arithmetic (MBA) obfuscation [43] has gained significant attention in recent years as an effective obfuscation technique. MBA obfuscation transforms simple expressions into complex expressions using a mixture of bitwise and arithmetic operations. For instance, an expression like $x - y$ can be transformed into $x \oplus y + 2(x \vee \neg y) + 2$ where $\oplus$ denotes bitwise exclusive-or, $\vee$ denotes bitwise or, and $\neg$ denotes bitwise not. Thanks to a solid theoretical foundation [43], MBA obfuscation enjoys arbitrarily many semantics-preserving transformations. In addition, it incurs a reasonable obfuscation cost and runtime overhead since only simple bitwise and arithmetic operations are added. Moreover, due to the high complexity of MBA expressions, it is resistant to both manual and automated reverse engineering. It is shown that off-the-shelf compiler optimizations [21] and symbolic reasoning tools [11, 17, 20, 30, 36, 40] cannot simplify MBA expressions because the bloated mixture of bitwise and arithmetic operations invalidates these techniques [33]. Thanks to these advantages, MBA obfuscation has been widely used in both commercial and academic obfuscators [1, 12, 15, 28, 32].

However, MBA obfuscation is not always used for good purposes. Malware authors use MBA obfuscation to protect their malware from reverse engineering. For example, MBA obfuscation has been used in malware compilation chains [9].

To protect against malware equipped with MBA obfuscation, various techniques for MBA deobfuscation have been proposed. These techniques include term rewriting [18], SAT solving [22], stochastic program synthesis [10, 16, 31], neural network inference [19], and algebraic methods [29, 34, 41].

Unfortunately, these techniques suffer from either of the following limitations.

- **Limited to a specific class of MBA expressions:** Some techniques [18, 22, 29, 34, 41] are limited to specific forms of MBA expressions, thereby failing to handle state-of-the-art MBA obfuscation techniques [15, 39]. They are limited to *linear* or *polynomial* MBA expressions with only the logical operators $\wedge, \vee, \neg, \oplus$ and the arithmetic operators $+, -, \times$. A polynomial MBA expression is a linear combination of a product of bitwise expressions of variables, and a linear MBA

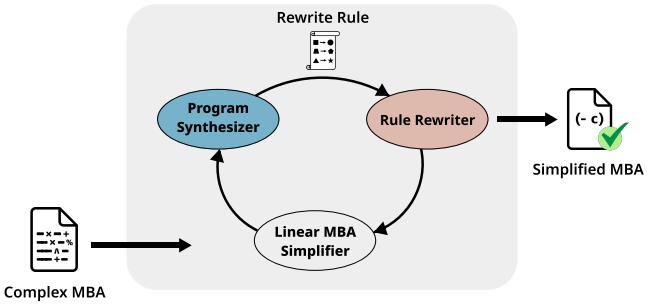


Figure 1: Overview of the ProMBA system.

expression is a polynomial MBA expression where each term is a bitwise expression of variables.

- **Lack of flexibility:** Because there are vastly many ways to obfuscate a program using MBA obfuscation, a deobfuscation technique should be flexible enough to handle a wide range of MBA transformations. However, some techniques [18, 19] are limited to specific transformations which are well-defined in the literature or observed in a set of samples.
- **Lack of scalability:** Because MBA obfuscation can greatly increase the size of an expression, a deobfuscation technique should be scalable enough to handle large MBA expressions. However, some techniques [18, 19, 22] are limited to small MBA expressions.
- **No guarantee of correctness:** It is desirable for a deobfuscation technique to guarantee the correctness of deobfuscation, especially when it is used in conjunction with program analysis techniques to detect malicious behaviors of malware. However, some techniques [10, 16, 19, 31] are not sound.

In this paper, we present an MBA deobfuscation system named ProMBA that overcomes the aforementioned limitations of existing techniques. Our method synergistically combines three powerful techniques: program synthesis [23], term rewriting [5], and an algebraic simplification method [34]. Figure 1 illustrates our approach.

(1) Given an MBA expression, we first simplify all linear MBA sub-expressions using an existing deobfuscator for linear MBA expressions. The existing deobfuscators based on algebraic methods [29, 34, 41] can efficiently simplify large linear MBA expressions. Therefore, any of such deobfuscators can be used in our approach. After this step, all linear MBA sub-expressions are simplified, but the resulting MBA expression may still contain non-linear MBA sub-expressions that can be simplified further.
(2) Given the simplified MBA expression, we aim to find an equivalent MBA expression that is smaller in size. This problem can be solved by program synthesis in theory. However, the synthesis problem is usually intractable because the search space is too large due to the significant size of the MBA expression. Therefore, we adopt a divide-and-conquer approach. We choose a sub-expression of the MBA expression and synthesize its equivalent counterpart of smaller size. By doing so, we reduce the search space and the complexity of the synthesis problem.
(3) From the chosen sub-expression and the synthesized one, we learn an MBA equivalence between them. This equivalence can be viewed as a transformation rule. We generalize the learned equivalence and add it to the set of rules to reduce their number and increase their flexibility.
(4) We apply the transformation rules to rewrite the MBA expression. The soundness and termination of the rewriting process are guaranteed by the theory of term rewriting [5]. After this step, the MBA expression may contain linear terms that can be further simplified by the linear MBA deobfuscator. We go back to Step 1 and repeat the process until the MBA expression cannot be simplified further.

The key novelty of our approach is that we perform *on-the-fly learning* of rules for simplification, and apply them to rewrite the MBA expression. This on-the-fly learning of transformation rules allows us to handle arbitrary (possibly non-linear and non-polynomial) MBA expressions obfuscated with diverse rules for obfuscation. This is in contrast to existing term rewriting-based deobfuscators that use a fixed set of rules [18], which limits their applicability to a specific type of obfuscation.

We have implemented our proposed method in a tool called ProMBA and evaluated it on 4011 MBA expressions obtained from various sources. Specifically, we used MBA expressions generated by state-of-the-art obfuscators Loki [39] and Tigress [2], as well as MBA expressions from the evaluation dataset of the state-of-the-art deobfuscators MBASolver [41] and Syntia [10]. Our method can successfully simplify 84% of the MBA expressions to their original expressions or even expressions that are smaller than the original ones. On the other hand, MBASolver and Syntia can simplify only 13% and 39% of the MBA expressions to their original expressions or smaller ones, respectively. In particular, MBASolver generates 36 times larger expressions on average than those generated by ProMBA, and Syntia does not guarantee the correctness of deobfuscation, occasionally generating incorrect expressions.

We summarize our contributions as follows:

- We propose a novel and versatile method for deobfuscating arbitrary (possibly non-linear and non-polynomial) MBA expressions using a combination of program synthesis and term rewriting. This enables the on-the-fly learning of arbitrary MBA equivalences that can be used to deobfuscate arbitrary MBA expressions effectively.
- Our method is implemented in a tool named ProMBA, and we evaluated it on MBA expressions generated by state-of-the-art obfuscators. Our evaluation results demonstrate that our approach outperforms the state-of-the-art deobfuscator, MBASolver [41].

The rest of the paper is organized as follows. Section 2 introduces preliminaries. Section 3 presents our approach in detail. Section 4 describes the implementation of our approach. Section 5 presents the evaluation results. Section 6 discusses the limitations of our approach. Section 7 discusses related work. Lastly, Section 8 concludes the paper.

2 BACKGROUND

In this section, we introduce the basic concepts of Mixed Boolean Arithmetic (MBA) obfuscation, syntax-guided program synthesis, and term rewriting.

2.1 MBA Obfuscation

We first introduce the basic concepts of MBA expressions from the work of Zhou et al. [43].

Boolean-Arithmetic Algebra. Boolean-arithmetic algebra is an algebraic system that consists of arithmetic operations and bitwise operations. Arithmetic operations include addition +, subtraction −, multiplication ×, division /, modulo operation %, left shift <<, logical right shift >>, and arithmetic right shift $>>^s$ operations. Bitwise operations include bitwise *and* ∧, *or* ∨, *not* ¬, and *exclusive-or* ⊕ operations. With n-bit 2's complement representation, arithmetic operations are in $\mathbb{Z}/2^n\mathbb{Z}$ and bitwise operations are in $\mathbb{B}^n$ where $\mathbb{B} = \{0, 1\}$. n is called the *dimension* of the algebra.

Polynomial and Linear MBA Expressions. With the Boolean-arithmetic algebra of dimension n and a positive integer t, any function $f : (\mathbb{B}^n)^t \to \mathbb{B}^n$ over t variables over $\mathbb{B}^n$ is an MBA expression. In particular, a function $f : (\mathbb{B}^n)^t \to \mathbb{B}^n$ of the form

$$\sum_{i \in I} a_i \Big(\prod_{j \in J_i} e_{i,j}(x_1, \cdots, x_t)\Big)$$

is a *polynomial Mixed Boolean-Arithmetic (MBA)* expression where a_i are constants in $\mathbb{Z}/2^n\mathbb{Z}$ and $e_{i,j}$ are bitwise expressions of variables $x_1, \cdots, x_t$ over $\mathbb{B}^n$, $I \subseteq \mathbb{Z}$, and $\forall i \in I$. $J_i \subseteq \mathbb{Z}$ are finite index sets. A constant can be viewed as a bitwise expression of the form $ae(x_1, \cdots, x_t)$ where a is a constant in $\mathbb{Z}/2^n\mathbb{Z}$ and $e(x_1, \cdots, x_t)$ is a bitwise expression of variables $x_1, \cdots, x_t$ over $\mathbb{B}^n$ that always evaluates to 1. For example, the MBA expression $8(x \vee y \vee z)^3((xy) \wedge x \vee t) + x + 9(x \vee y)yz^3$ is a polynomial MBA expression.

A linear MBA expression is a polynomial MBA expression of the form

$$\sum_{i \in I} a_i e_i(x_1, \cdots, x_t).$$

For example, the MBA expression $x + y - 2(x \vee y) + 1$ is a linear MBA expression.

Any MBA expression not a polynomial MBA expression is a *non-polynomial* MBA expression. A non-polynomial MBA expression may contain arithmetic operations other than addition, subtraction, multiplication, or bitwise expressions with constants. For example, the MBA expression $(x >> 1) + y - 3(x \wedge 2)$ is a non-polynomial MBA expression.

The work by Zhou et al. [43] provides the theoretical foundations for MBA obfuscation. It is shown that every operation in Boolean-arithmetic algebra can be expressed as a high-degree polynomial MBA expression of multiple terms over arbitrarily many variables.

2.2 Term Rewriting

Next, we introduce the basic concepts of term rewriting from the work of Baader and Nipkow [5].

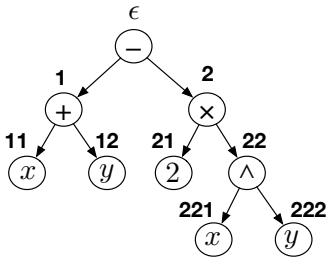


Figure 2: Tree representation of the term $s = (x + y) - 2(x \wedge y)$ and its positions.

Term. Given a set $\mathcal{F}$ of function symbols, where each $f \in \mathcal{F}$ is associated with a natural number n called the *arity* of f (denoted $arity(f)$) and a set $\mathcal{X}$ of variables, $\mathcal{T}_{\mathcal{F},\mathcal{X}}$ is the set of terms built from functions in $\mathcal{F}$ and variables in $\mathcal{X}$. In other words, $\mathcal{X} \subseteq \mathcal{T}_{\mathcal{F},\mathcal{X}}$ and $\forall f \in \mathcal{F}$. $f(t_1, \cdots, t_n) \subseteq \mathcal{T}_{\mathcal{F},\mathcal{X}}$ where $n = arity(f)$ and $t_1, \cdots, t_n \in \mathcal{T}_{\mathcal{F},\mathcal{X}}$. Function symbols of 0-arity are *constants*. We will often denote a function symbol of arity 2 as an infix operator and $Var(s)$ and $Const(s)$ as the set of variables and constants in term s respectively. MBA expressions in the Boolean-arithmetic algebra of dimension n are terms over the arithmetic and bitwise operations and variables and constants over $\mathbb{B}^n$.

The structure of a term can be illustrated by a tree.

Example 2.1. Consider an MBA expression $s = (x + y) - 2(x \wedge y)$ over the Boolean-arithmetic algebra of dimension n. s is a term over $\mathcal{T}_{\mathcal{F},\mathcal{X}}$ where $\mathcal{F}$ is the set of arithmetic and bitwise operations and constants over $\mathbb{B}^n$, and $\mathcal{X} = \{x, y\}$. The term s can be represented by the tree in Figure 2.

Position. The set of positions of term s (denoted $Pos(s)$) of strings over the alphabet of natural numbers is defined recursively as follows:

- if s is a variable or a constant, then $Pos(s) = \{\epsilon\}$;
- if $s = f(t_1, \cdots, t_n)$, then $Pos(s) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(t_i)\}$.

The position ϵ is called the *root* position of s. For $p \in Pos(s)$, the subterm of s at position p (denoted $s|_p$) is defined by (i) $s|_\epsilon = s$; (ii) $f(t_1, \cdots, t_n)|_{iq} = t_i|_q$. The number of positions of s is denoted $|s|$. The height of s (denoted Height(s)) is defined as $\max_{p \in Pos(s)} |p|$. We denote $s[p \leftarrow t]$ as the term obtained by replacing the subterm of s at position p with t.

Example 2.2. In Figure 2, the positions of s are ϵ, 1, 2, 11, 12, 21, 22, 221, 222, which are written on top of the corresponding nodes. The size of s is 9 because we have 9 positions, and the height of s is 3 because the longest position 222 has length 3. The subterm of s at position 22 is $s\,|_{22} = x \wedge y$ and the term obtained by replacing the subterm of s at position 22 with z is $s[22 \leftarrow z] = (x + y) - 2z$.

Substitution. A substitution σ is a function from $\mathcal{X}$ to $\mathcal{T}_{\mathcal{F},\mathcal{X}}$. The set of substitutions over $\mathcal{T}_{\mathcal{F},\mathcal{X}}$ is denoted $Sub(\mathcal{T}_{\mathcal{F},\mathcal{X}})$. Every substitution σ can be immediately extended to a function σ' from $\mathcal{T}_{\mathcal{F},\mathcal{X}}$ to $\mathcal{T}_{\mathcal{F},\mathcal{X}}$ as follows: (i) $\sigma'(x) = \sigma(x)$ for all $x \in \mathcal{X}$; (ii) $\sigma'(f(t_1, \cdots, t_n)) =$

$f(\sigma'(t_1), \cdots, \sigma'(t_n))$ for all $f \in \mathcal{F}$ and $t_1, \cdots, t_n \in \mathcal{T}_{\mathcal{F},\mathcal{X}}$. From now on, we will use σ to denote σ' with a slight abuse of notation.

Example 2.3. Let $\sigma = \{x \mapsto 1, y \mapsto z + 1\}$. Then, for the term s in Example 2.1, $\sigma(s) = (1 + (z + 1)) - 2(1 \wedge (z + 1))$.

Term Rewriting. A rewrite rule is a pair (s, t) (written $s \to t$) where $s, t \in \mathcal{T}_{\mathcal{F},\mathcal{X}}$, s is not a variable, and $Var(t) \subseteq Var(s)$. A term rewriting system consists of a set E of rewrite rules over $\mathcal{T}_{\mathcal{F},\mathcal{X}}$. The rewrite relation $\to_E$ is defined as follows:

$$s \to_E t \Leftrightarrow \exists l \to r \in E, p \in Pos(s), \sigma \in Sub(\mathcal{T}_{\mathcal{F},\mathcal{X}}). \\ s|_p = \sigma(l), t = s[p \leftarrow \sigma(r)].$$

Example 2.4. Let $E = \{a + b \to (a \oplus b) + 2(a \wedge b), (a + b) - c \to a + (b - c), a - a \to 0, a + 0 \to a\}$. Then, we can rewrite the term s in Example 2.1 as follows:

$$\begin{aligned} s &\to_E ((x \oplus y) + 2(x \wedge y)) - 2(x \wedge y) \\ &\quad \text{(by using } a + b \to (a \oplus b) + 2(a \wedge b) \text{ and } \sigma = \{a \mapsto x, b \mapsto y\}) \\ &\to_E (x \oplus y) + (2(x \wedge y) - 2(x \wedge y)) \\ &\quad \text{(by using } (a + b) - c \to a + (b - c)) \text{ and} \\ &\quad \sigma = \{a \mapsto (x \oplus y), b \mapsto 2(x \wedge y), c \mapsto 2(x \wedge y)\}) \\ &\to_E (x \oplus y) + 0 \\ &\quad \text{(by using } a - a \to 0 \text{ and } \sigma = \{a \mapsto 2(x \wedge y)\}) \\ &\to_E (x \oplus y) \\ &\quad \text{(by using } a + 0 \to a \text{ and } \sigma = \{a \mapsto (x \oplus y)\}). \end{aligned}$$

2.3 Syntax-Guided Synthesis

Lastly, we introduce the basic concepts of syntax-guided synthesis [3].

Context-Free Grammar. A context-free grammar is a tuple $G = \langle N, \Sigma, S, \delta \rangle$ where N is a finite set of non-terminals, Σ is a finite set of terminals, $S \in N$ is the start non-terminal, and δ is a set of production rules of the form $A_0 \to f(A_1, \cdots, A_n)$ where $A_0, \cdots, A_n \in N$, $f \in \Sigma$, and $arity(f) = n$. Given a term $t \in \mathcal{T}_{\mathcal{F},\mathcal{X}}$, applying a production rule $A \to \beta \in \delta$ to t replaces an occurrence of a non-terminal A in t with β. All terms that can be generated by G are called the *language* of G and denoted by $L(G)$[1].

Syntax-Guided Synthesis. The syntax-guided synthesis (SyGuS) problem [3] is a tuple $\langle G, \Phi \rangle$. The goal is to find a term $p \in L(G)$ that satisfies the specification Φ in a decidable theory $\mathcal{T}$. We assume that each term p is with a deterministic semantics (denoted $[\![p]\!]$). The specification Φ is a formula $\Phi(x, p)$ that relates the input x and the output $[\![p]\!](x)$ of p. The goal is to find a term $p \in L(G)$ such that $\forall x.\ \Phi(x, [\![p]\!](x))$ is valid modulo the decidable theory $\mathcal{T}$. We often call G the syntactic specification and $\forall x.\ \Phi(x, [\![p]\!](x))$ the semantic specification.

Simplifying an MBA expression can be formulated as a syntax-guided synthesis problem where the syntactic specification is a grammar of MBA expressions and the semantic specification is a formula representing the equivalence between the original MBA expression and the simplified one.

[1]The grammar is actually called regular tree grammar. However, we stick to the more familiar name context-free grammar for simplicity of presentation.

Example 2.5. Suppose that we want to simplify the MBA expression s in Example 2.1. This problem can be formulated as a syntax-guided synthesis problem The syntactic specification is the grammar $G = \langle \{S, V, C\}, \mathcal{F} \cup \mathcal{X}, S, \delta \rangle$ where δ comprises the following production rules:

$$\begin{array}{lcl} S & \to & V \mid C \mid S + S \mid S \oplus S \mid S \wedge S \mid S - S \mid \cdots \\ V & \to & x \mid y \\ C & \to & 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

The non-terminal V represents the set of variables $\mathcal{X}$. The non-terminal C represents a set of constants. The start non-terminal S represents the set of all MBA expressions involving the variables and constants.

The semantic specification is the following formula defined in the theory of bit-vectors of fixed-width n:

$$\forall x, y \in \mathbb{B}^n.\ [\![p]\!](x, y) = [\![s]\!](x, y)$$

where p is the term to be synthesized. Given this problem, an existing SyGuS solver [42] can find the minimal solution $p = x \oplus y$ less than 0.1 seconds.

Among the existing SyGuS solvers based on various synthesis strategies, enumerative solvers [4, 16, 26, 42] are the most suitable for our purpose. That is because they enumerate all the candidate solutions in a certain order guaranteeing that the first solution found is minimal. On the other hand, other strategies such as stochastic search [38] or probabilistic model-based search [7, 27] may find a solution which is not minimal.

3 OUR APPROACH

This section describes our approach to the problem of simplifying MBA expressions.

In theory, by using a sound and complete SyGuS solver as described in Section 2, we can find a minimal sub-expression that is semantically equivalent to an original MBA expression. However, in practice, because MBA expressions are usually very large, this method is not feasible due to the scalability issue of SyGuS solvers.

To address this scalability issue, we adopt a divide-and-conquer approach that simplifies a given MBA expression by recursively simplifying its sub-expressions. We divide the MBA expression into multiple sub-expressions, simplify each sub-expression, and then combine the simplified sub-expressions to obtain a simplified MBA expression. This process is repeated until no further simplification is possible.

3.1 The Overall Algorithm

The pseudocode of our algorithm is shown in Algorithm 1. The algorithm is inspired by the prior work on optimizing homomorphic encryption circuits using syntax-guided synthesis [25]. Though the overall structure of the algorithm is similar to the prior work, the key difference between our algorithm and the prior work [25] lies in the learning process. While the prior work relies on offline learning, where simplification rules are learned from a set of training examples and subsequently applied to target expressions, our algorithm performs on-the-fly learning of rules. As will be demonstrated in Section 5.4, this on-the-fly learning approach is more effective than the offline learning approach.

Algorithm 1 The on-the-fly learning-based MBA Deobfuscation

Require: P: obfuscated MBA expression of n-bit input variables
Ensure: p: simplified MBA expression
1: $p \leftarrow$ SimplifyLinearMBAs(P)
2: $h \leftarrow 1$ ▷ Height of sub-expressions to simplify
3: $E \leftarrow \emptyset$ ▷ Rule set
4: **while** $h \leq$ Height(p) **do**
5: $W \leftarrow Pos(p)$ ▷ Positions of sub-expressions to simplify
6: **while** $W \neq \emptyset$ **do**
7: remove a pos from W
8: $\langle r, \sigma \rangle \leftarrow$ ChooseSubExpr($p\,|_{pos}, h$)
9: $r' \leftarrow$ Synthesize(r)
10: **if** $r' = \bot$ **then**
11: **continue**
12: **else if** $|r'| < |r|$ **then**
13: $E \leftarrow E \cup \{$GeneralizeRule($r \rightarrow r'$)$\}$
14: $p \leftarrow p[pos \leftarrow \sigma(r')]$
15: $p \leftarrow$ ApplyRule(E, p)
16: $p \leftarrow$ SimplifyLinearMBAs(p)
17: $W \leftarrow Pos(p)$
18: **end if**
19: **end while**
20: $h \leftarrow h + 1$
21: **end while**
22: **return** p

The algorithm takes as input an obfuscated MBA expression P and returns a simplified MBA expression p. At each iteration, the algorithm chooses a sub-expression e of a bounded height, and attempts to synthesize a simpler sub-expression e' that is equivalent to e. Then, the algorithm applies the rule $e \rightarrow e'$ to the other sub-expressions to take advantage of the simplification. The algorithm terminates when no further simplification is possible.

The algorithm first identifies linear MBA sub-expressions in P and simplifies them (line 1) using an off-the-shelf deobfuscator for linear MBA expressions [29, 34, 41]. After this step, all the linear MBA sub-expressions in P are simplified, but the other non-linear MBA sub-expressions may be further reducible. Then, it initializes the maximum height h of sub-expressions to consider to be 1, the rule set E to be empty, and the set of positions W of sub-expressions to be the set of all positions of P (lines 2–5).

At each iteration of the inner loop, Given a position pos of a sub-expression to consider (line 7), the ChooseSubExpr function chooses a sub-expression r of height at most h at the position pos (line 8). The ChooseSubExpr function substitutes some sub-expressions with fresh variables and returns the resulting sub-expression r and the substitution σ. Section 3.2 describes how to implement the ChooseSubExpr function.

Example 3.1. Consider the following MBA expression p:

$$p = (((x_2 \times x_3) \wedge (\neg x_1)) + x_2) - ((x_1 \vee (x_2 \times x_3)) \wedge (x_2 - x_1))$$

whose height is 4. Suppose that the height h is 2 and the selected position is the root position. Then, given the subterm $p\,|_\epsilon$ and h, the ChooseSubExpr function returns the sub-expression

$$r = (t_1 + x_2) - (t_2 \wedge t_3)$$

along with a substitution

$$\sigma = \{t_1 \mapsto x_2 \times x_3, t_2 \mapsto x_1 \vee (x_2 \times x_3), t_3 \mapsto x_2 - x_1\}.$$

Note that the height of r is 2 and t_1, t_2 and t_3 are fresh variables used to abstract away the corresponding sub-expressions to make the height of r at most h.

Next, the algorithm invokes a SyGuS solver to synthesize a simpler sub-expression r' that is equivalent to the sub-expression r (line 9). If the synthesis fails, the algorithm continues to another sub-expression (line 10). If the synthesis succeeds and the size of the synthesized sub-expression r' is smaller than the size of the original sub-expression r (line 12), then it means we can obtain a rewrite rule $r \rightarrow r'$.

To maximize the chance of applying the rule $r \rightarrow r'$ to other sub-expressions, the algorithm generalizes the rule $r \rightarrow r'$ by replacing sub-expressions that appear both in r and r' with variables (line 13), and adds the generalized rule to the rule set E. After replacing the old sub-expression r with the new sub-expression r' in p (line 14), it applies the rules in E to other sub-expressions of p to which the rule $r \rightarrow r'$ can be applied (line 15).

After this process, existing non-linear MBA sub-expressions may become linear MBA sub-expressions, and thus the algorithm simplifies them again (line 16) using a linear MBA deobfuscator.

The algorithm then updates the set W to be the set of all positions of the newly obtained expression p (line 17), and repeats the inner loop.

Because the set W is updated only when the expression p is changed, the inner loop terminates when no further simplification is possible (line 19). Then, the algorithm increases the height h of sub-expressions to consider by 1 (line 20), and repeats the inner loop again until the height h of sub-expressions to consider exceeds the height of p (line 21). Finally, the algorithm returns the obtained expression p (line 22).

The reason why we increase the height h of sub-expressions to consider is that the algorithm may fail to simplify a sub-expression of a limited height if reducible parts of the sub-expression are abstracted by fresh variables. However, the algorithm may be able to simplify it if the height of the sub-expression is increased.

Example 3.2. Consider the MBA expression p and its sub-expression r in Example 3.1. If we try to synthesize a simpler sub-expression that is equivalent to r of which the height is 2, the synthesis fails to find a simpler sub-expression. However, if we try to synthesize a simpler expression equivalent to p in Example 3.1 of which the height is 4, the synthesis succeeds to find a simpler sub-expression $x_1 \vee ((x_2 \times x_3) \vee (x_2 - x_1))$ and obtaining the following rewrite rule:

$$(((x_2 \times x_3) \wedge (\neg x_1)) + x_2) - ((x_1 \vee (x_2 \times x_3)) \wedge (x_2 - x_1)) \rightarrow x_1 \vee ((x_2 \times x_3) \vee (x_2 - x_1))$$

Our algorithm is correct and terminating in the following sense.

Theorem 3.3. *Algorithm 1 eventually returns a simplified MBA expression p that is equivalent to the input MBA expression P.*

3.2 The ChooseSubExpr Procedure

The ChooseSubExpr function depicted in Algorithm 2 chooses a sub-expression r of height at most h of a given expression e. If the

Algorithm 2 ChooseSubExpr

Require: e: input MBA expression
Require: h: maximum height of sub-expressions to consider
Ensure: a sub-expression of e
Ensure: a substitution from variables to expressions
1: **if** Height$(e) \leq h$ **then**
2: **return** $\langle e, \emptyset \rangle$
3: **else if** $h = 0$ **then**
4: $x \leftarrow$a new fresh variable
5: **return** $\langle x, \{x \mapsto e\} \rangle$
6: **else**
7: $\sigma \leftarrow \emptyset$
8: **for** $i \leftarrow 1$ to number of children of e **do**
9: $\langle r, \sigma' \rangle \leftarrow$ ChooseSubExpr$(e\,|_i, h-1)$
10: $e \leftarrow e[i \leftarrow r]$
11: $\sigma \leftarrow \sigma \cup \sigma'$
12: **end for**
13: **return** $\langle e, \sigma \rangle$
14: **end if**

given expression e has height at most h, the function returns the expression e and an empty substitution (line 2). If the height h is 0, the function returns a fresh variable x and a substitution $\{x \mapsto e\}$ (lines 4–5). Otherwise, the function chooses a sub-expression r of height at most $h-1$ for each child of e (line 8). During this process, the function maintains a substitution σ from variables to expressions (line 7), which is initially empty. As the children of e are processed, the function updates the expression e by replacing the i-th child of e with the sub-expression r of the limited height (line 10), and updates the substitution σ by conjoining σ' (line 11). Finally, the function returns the expression e and the substitution σ (line 13).

3.3 The Synthesize Procedure

Given a sub-expression e where $Var(e) = \{x_1, \ldots, x_t\}$ and $Const(e) = \{c_1, \ldots, c_m\}$, the Synthesize procedure aims to find a new sub-expression e' that is semantically equivalent to e. This problem is formulated as a SyGuS problem as follows. The target expression is e' and the background theory is the theory of bit-vectors of fixed-width n. The syntactic specification for e' is

$$\begin{array}{lcl} S & \to & V \mid C \mid S+S \mid S-S \mid S \times S \mid \cdots \\ V & \to & x_1 \mid x_2 \mid \cdots \mid x_t \\ C & \to & 0 \mid 1 \mid c_1 \mid c_2 \mid \cdots \mid c_m \end{array}$$

where S is the start symbol, and V and C are non-terminals for variables and constants, respectively. We include all the logical and arithmetic operators in the theory of bit-vectors in the production rules for S. Precisely, the operators we use are (in SMT-LIB [8]) `bvadd`, `bvsub`, `bvmul`, `bvdiv`, `bvudiv`, `bvrem`, `bvurem`, `bvshl`, `bvashr`, `bvlshr`, `bvneg`, `bvand`, `bvor`, `bvxor`, and `bvnot`. The semantic specification for e' is

$$\forall x_1, x_2, \cdots, x_t \in \mathbb{B}^n.\ e(x_1, \cdots, x_t) = e'(x_1, \cdots, x_t).$$

The Synthesize procedure encodes this SyGuS problem in the SYNTH-LIB format [3] and invokes a SyGuS solver to solve the problem.

Note that the SyGuS problem for the Synthesize procedure is decidable because SyGuS problems with the theory of quantifier-free bit-vectors without bitstring concatenation are decidable since the domain is finite [14]. Furthermore, all enumerative SyGuS solvers including EUSolver [4], Duet [26], and SimbaS [34] are sound and complete in that they always return a solution if there exists one. However, these solvers are not guaranteed to terminate within a given timeout. If the solver returns a solution within the given timeout, the procedure returns the solution as the new sub-expression e'. Otherwise, the procedure returns $\perp$.

3.4 The GeneralizeRule Procedure

The GeneralizeRule procedure generalizes a rule $l \to r$ in a similar way to the Normalize procedure in the work by Lee et al (Section 4.2.4 in [25]). It first identifies sub-expressions that are common to both l and r. Then, it replaces the common sub-expressions with fresh variables.

Example 3.4. Consider the rewrite rule in Example 3.2. The common sub-expressions that are shared by both left-hand side and right-hand side are $x_2 \times x_3$ and $x_2 - x_1$. If we replace these common sub-expressions with fresh variables y_1 and y_2, respectively, the rule becomes

$$((y_1 \wedge (\neg x_1)) + x_2) - ((x_1 \vee y_1) \wedge y_2) \to x_1 \vee (y_1 \vee y_2).$$

Because the replacement of common sub-expressions with fresh variables may change the semantics of the rule, it checks whether the rule is still valid after the replacement using an SMT solver. If the rule is valid, the procedure returns the generalized rule. Otherwise, the procedure returns $\perp$.

Example 3.5. Suppose we are given the following rule

$$(a \wedge b) \wedge (b \wedge a) \to (a \wedge b) \wedge b.$$

If we replace the common sub-expression $(a \wedge b)$ with a fresh variable x, the rule becomes

$$x \wedge (b \wedge a) \to x \wedge b.$$

This rule is not valid because the left-hand side of the rule is not equivalent to the right-hand side.

3.5 The ApplyRule Procedure

Given a set of rules E and an expression e, the ApplyRule procedure applies rules in the set E to e. It is based on the following term rewriting system whose rewrite relation is defined by the set of rules E.

$$\begin{array}{l} s \to_E t \Leftrightarrow \exists l \to r \in E, p \in Pos(s), \sigma \in Sub(\mathcal{T}_{\mathcal{F},\mathcal{X}}). \\ \qquad s|_p = \sigma(l), |\sigma(l)| > |\sigma(r)|, t = s[p \leftarrow \sigma(r)]. \end{array}$$

Because our goal is to reduce the size of the expression e, we admit only those rules that reduce the size of the expression e. This is why we require that $|\sigma(l)| > |\sigma(r)|$ in the above definition.

The ApplyRule procedure returns a new expression e' such that $e \to_E^* e'$, where $\to_E^*$ is the reflexive transitive closure of $\to_E$.

Since the rewrite relation $\to_E$ is not confluent in general[2], the ApplyRule procedure may return different expressions for the same

[2]A term rewriting system is confluent if for any terms s, t_1, t_2 such that $s \to t_1$ and $s \to t_2$, there exists a term t such that $t_1 \to^* t$ and $t_2 \to^* t$.

input expression depending on the order of rules in E. To avoid this non-determinism, the ApplyRule procedure applies rules in E in a fixed order. In case of multiple rules that can be applied to the same sub-expression, the procedure applies the rule that is learned least recently.

3.6 Optimizations

We apply several optimizations to the algorithm to improve the simplification performance.

Using an Initial Set of Rules. To expedite the simplification process, instead of starting from the empty set of rules (line 5 in Algorithm 1), we start from a set of simple rules. For example, the initial set of rules include basic arithmetic properties such as $x + 0 \rightarrow x$, $x \times 1 \rightarrow x$, $x - x \rightarrow 0$ and $x \times 0 \rightarrow 0$. Though such simple rules can be learned by the algorithm, we can save the time to learn them by providing them as the initial set of rules.

Optimizations for the ChooseSubExpr **Procedure.** In the ChooseSubExpr procedure, we use the same fresh variable for the same sub-expression to avoid generating redundant fresh variables. This optimization is necessary not to miss the opportunity to synthesize a smaller expression.

Example 3.6. Suppose the procedure is given the following expression e along with the maximum height $h = 1$.

$$(x_1 + x_2) - (x_1 + x_2).$$

According to the ChooseSubExpr procedure as described in Algorithm 8, it first recursively calls itself with the left sub-expression $(x_1 + x_2)$ and the maximum height $h = 0$. Then, it returns a fresh variable y_1 for the sub-expression $(x_1 + x_2)$. Next, it recursively calls itself with the right sub-expression (x_1+x_2) and the maximum height $h = 0$. Then, it returns another fresh variable y_2 for the sub-expression $(x_1 + x_2)$. The resulting expression and the substitution are $y_1 - y_2$ and $\{y_1 \mapsto (x_1+x_2), y_2 \mapsto (x_1+x_2)\}$, respectively. The syntheizer cannot synthesize a smaller expression that is equivalent to the expression $y_1 - y_2$ as it cannot recognize that y_1 and y_2 are the same sub-expression. If we use the same fresh variable for the same sub-expression, the procedure returns the same fresh variable y_1 for the sub-expression $(x_1 + x_2)$. Then, the resulting expression and the substitution would be $y_1 - y_1$ and $\{y_1 \mapsto (x_1 + x_2)\}$, respectively. The synthesizer can synthesize a smaller expression 0 to replace $y_1 - y_1$.

To avoid generating different fresh variables for the same sub-expression, we use a hash table to store the mapping from sub-expressions to fresh variables and check whether the same sub-expression has already been assigned a fresh variable. If so, we use the same fresh variable for the sub-expression.

Optimization for the Synthesize **Procedure.** If the Synthesize procedure fails to synthesize an equivalent expression for a given expression e and e can be viewed as an expression of the form $e_1 \odot e_2 \odot \cdots \odot e_n$, where $\odot$ is an associative and commutative operator and $e_1, \ldots, e_n$ are MBA expressions, then we attempt to synthesize an equivalent but smaller expression for every possible pair of sub-expressions $e_i \odot e_j$.

To understand the necessity of this optimization, consider the following example.

Example 3.7. Suppose the Synthesize procedure at line 9 in Algorithm 1 fails to synthesize a smaller expression that is equivalent to the following given expression r:

$$-5y + (9(\neg x) + (5(x \wedge y) + xy)).$$

The failure of the Synthesize procedure does not necessarily mean that the given expression is already the smallest expression. Indeed, there is a sub-expression $e = -5y + 5(x \wedge y)$ that can be simplified into $-5(y \wedge (\neg x))$.

Not to miss such a simplification opportunity, we can treat associative and commutative binary operators as n-ary operators and regard every possible combination of operands of the n-ary operator as a potential sub-expression to be simplified. However, this may result in a large number of sub-expressions for which we need to synthesize equivalent but smaller expressions. To reduce the number of sub-expressions, we consider only pairs of operands of the n-ary operator.

For the expression r, we attempt to synthesize an equivalent but smaller expression for each of the following sub-expressions: $-5y+5(x\wedge y), -5y+xy, -5y+9(\neg x), 5(x\wedge y)+xy, 5(x\wedge y)+9(\neg x)$, and $xy + 9(\neg x)$. Then, the synthesizer can simplify the sub-expression $-5y + 5(x \wedge y)$ into $-5(y \wedge (\neg x))$. Therefore, we can replace the sub-expression $-5y + 5(x \wedge y)$ with $-5(y \wedge (\neg x))$ to obtain the following expression:

$$-5(y \wedge (\neg x)) + (xy + (9(\neg x))).$$

If the synthesizer fails to synthesize an equivalent but smaller expression for every pair of sub-expressions, the Synthesize procedure returns $\perp$ to indicate that the expression cannot be simplified further.

4 IMPLEMENTATION

We have implemented our approach as a tool named ProMBA[3]. ProMBA is written in OCaml and consists of 2800 lines of code. As depicted in Algorithm 1, our ProMBA system consists of three major components: (1) a linear MBA deobfuscation engine, (2) a SyGuS solver, and (3) a verifier for checking the correctness of generalized rewrite rules.

Coincidentally, the linear MBA deobfuscation engine and the SyGuS solver used in ProMBA have the same name, Simba. To avoid confusion, we call the linear MBA deobfuscation engine SimbaD and and the SyGuS solver SimbaS in the rest of this paper.

The linear MBA deobfuscation engine SimbaD [34] is based on algebraic simplification. Given a linear MBA expression, SimbaD obtains input-output examples by evaluating the expression with all possible combinations of zeros and ones considering all input variables as 1-bit variables. Then, from the input-output examples, it constructs a linear combination of terms where each term is a conjunction of input variables. After applying several simplification rules, it obtains a simpler equivalent expression. This process is quite effective for linear MBA expressions, but it is not applicable to non-linear MBA expressions. Therefore, we can use SimbaD only for linear MBA expressions.

[3]Tool is available at https://github.com/astean1001/ProMBA

We chose SimbaD among the existing linear MBA deobfuscation tools because it is the most effective tool in terms of success rate and running time as shown in the previous work [34].

For the SyGuS solver, we use the SimbaS SyGuS solver [42]. SimbaS adopts an enumerative search strategy that synergistically combines the top-down and bottom-up enumerative search strategies. Most of all, SimbaS is specialized for the synthesis of bit-vector expressions by using a highly precise static analysis for bit-vectors to prune the search space. Therefore, according to the experiment in [42], SimbaS is the most efficient SyGuS solver for synthesizing bit-vector expressions. For each SyGuS query, we set the timeout to 20 seconds.

Lastly, for verifying the correctness of generalized rewrite rules, we use the Z3 SMT solver [17].

5 EVALUATION

This section evaluates our ProMBA system to answer the following research questions:

- **RQ1:** How effective is ProMBA for deobfuscating MBA expressions in terms of success rate and running time?
- **RQ2:** How does ProMBA compare with the state-of-the-art MBA deobfuscation tool?
- **RQ3:** What is the impact of the underlying synthesis engine on the performance of ProMBA?
- **RQ4:** How effective is ProMBA's on-the-fly learning of rewrite rules?
- **RQ5:** How effective is ProMBA's optimization techniques described in Section 3.6?

All of our experiments were conducted on a Linux machine with Intel Xeon 2.6GHz CPUs and 256GB of memory. Furthermore, we use $n = 64$ bits in all experiments (i.e., all input variables of MBA expressions are 64-bit variables).

5.1 Experimental Setup

Datasets. We aim to evaluate ProMBA using a large number of *non-linear* MBA expressions. Note that we do not consider linear MBA expressions in our evaluation. ProMBA deobfuscates any linear MBA sub-expressions using the state-of-the-art linear MBA deobfuscation tool SimbaD [34], thereby focusing on more complicated non-linear MBA sub-expressions. Since applying ProMBA on linear MBA expressions is simply equivalent to applying SimbaD, we exclude existing datasets containing only linear MBA expressions, such as the Neureduce dataset [19].

Table 1 summarizes the characteristics of the datasets. The *ground truth* (i.e., the original expression before obfuscation) is available for every MBA expression in the datasets. We collect **4011** non-linear MBA expressions from the following sources:

- **MBA-Solver dataset**: This dataset comprises **2011** MBA deobfuscation problems used to evaluate MBASolver [41]. The dataset contains both non-linear polynomial and non-polynomial MBA expressions.
- **QSynth dataset**: This dataset comprises **500** MBA deobfuscation problems used to evaluate QSynth [16]. The 500 MBA expressions are obfuscated with the *EncodeArithmetic* scheme [15] in the Tigress obfuscator [2].
- **Loki dataset**: This dataset comprises **1500** MBA deobfuscation problems. All of the MBA expressions are obfuscated with the recursive and randomized expression rewriting method used in the Loki obfuscator [39][4]. Among them, 1000 MBA expressions were used to evaluate the obfuscation resilience of Loki against existing four MBA deobfuscation tools (MBABlast, SSPAM, Arbyo, and NeuReduce). To meet the various requirements of the four tools, these 1000 MBA expressions are in a limited form that only contains addition, subtraction, logical and, logical or, and logical xor operations without any constants. To evaluate the effectiveness of ProMBA against the full-fledged MBA obfuscation, we additionally generate 500 MBA expressions that contain constants and all of the arithmetic and logical operations by using the full-fledged MBA expression generator in Loki.

Baseline MBA Deobfuscator. We compare ProMBA to two state-of-the-art MBA deobfuscation tools: MBASolver [41], which guarantees the correctness of deobfuscation but can only deobfuscate a limited class of non-linear MBA expressions, and Syntia [10], which can deobfuscate a wide range of non-linear MBA expressions but does not guarantee the correctness of deobfuscation.

MBASolver [41] is specialized for deobfuscating linear MBA expressions, but it can also deobfuscate a limited class of non-linear MBA expressions that only contain addition, subtraction, multiplication, logical and, logical or, and logical not operations. MBASolver transforms bitwise sub-expressions in an MBA expression into linear ones and uses various heuristic rules to further simplify the expression. The rules include replacing common sub-expressions with variables, using a pre-computed mapping table for commonly used MBA expressions, and the common math rules such as the distributive law and additive cancellation. MBASolver requires the user to provide additional information about a target MBA expression if it contains a non-polynomial sub-expression. It replaces the non-polynomial sub-expression with a fresh variable, performs deobfuscation on the substituted polynomial expression, and then reverts the substitution to obtain the final result. The user provides the information about which sub-expression to substitute. We provide the largest and most frequently appearing non-polynomial sub-expression. We let MBASolver perform deobfuscation on every sub-expression until there are no more sub-expressions that can be reduced by it.

Syntia [10] is capable of deobfuscating any arbitrary MBA expression by synthesizing a deobfuscated expression from finitely many input-output examples. The correctness of the deobfuscated result is not guaranteed. Syntia's algorithm is based on Monte Carlo Tree Search (MCTS), which is a heuristic search algorithm with a random component. For each deobfuscation task, we execute Syntia 30 times, each with 10 randomly selected input-output examples, and select the best outcome as the final result.

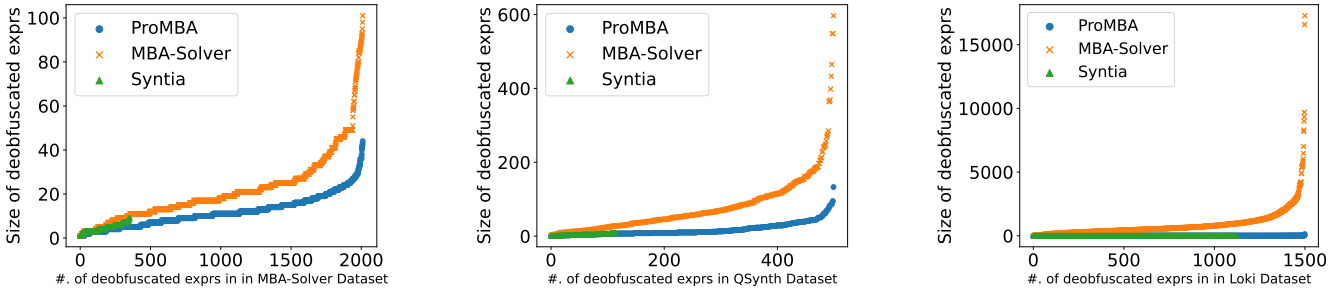
There are other MBA deobfuscation tools which we do not compare against due to the following reasons.

- SimbaD [34], MBABlast [29]: These tools only target linear MBA expressions.

[4]The recursive expression rewriting bound, which is the maximum number of subsequent applications of rewriting rules to an expression, is set to 30.

Table 1: Characteristics of the datasets. "Type" indicates the number of (non-linear) polynomial and non-polynomial MBA expressions. "#Vars", "Size", and "Original Size" report the number of input variables, the size of the obfuscated MBA expressions, and the size of the original expressions before obfuscation (i.e., ground truth), respectively.

Datasets	Type		#Vars (Min/Max/Avg)	Size (Min/Max/Avg)	Original Size (Min/Max/Avg)
MBA-Solver	Poly: 1008	Non-poly: 1003	1 / 4 / 2.79	4 / 606 / 191.89	1 / 32 / 13.17
QSynth	Poly: 0	Non-poly: 500	1 / 3 / 2.32	13 / 2593 / 245.8	6 / 21 / 13.4
Loki	Poly: 3	Non-poly: 1497	1 / 4 / 2.16	17 / 34553 / 1568.53	3 / 7 / 3.65
Total	Poly: 1011	Non-poly: 3000	1 / 4 / 2.5	4 / 34553 / 713.44	1 / 32 / 9.64



(a) Size of deobfuscated expressions (MBASolver) **(b) Size of deobfuscated expressions (QSynth)** **(c) Size of deobfuscated expressions (Loki)**

Figure 3: The distribution of the size of deobfuscated expressions for each dataset. We compare the results of ProMBA with those of MBASolver and Syntia.

- SSPAM [18] and NeuReduce [19][5] : These tools have been shown ineffective in deobfuscating large non-linear MBA expressions [39].
- QSynth [16] and Xyntia [31]: The core algorithms of these tools are similar to Syntia, which is synthesis from random input-output examples. Therefore, we believe that comparing Syntia with ProMBA suffices for our evaluation.
- Gamba [35]: The tool was not publicly available at the time of writing. We qualitatively compare ProMBA with Gamba in Section 7.

Evaluation Metrics. We evaluate ProMBA in terms of the following metrics:

- **Size of the deobfuscated expression**: Considering expressions as abstract syntax trees (ASTs), we measure the size of deobfuscated expressions by the number of AST nodes.
- **Success Rate**: We measure the *success rate* of the tools. We consider a deobfuscation task to be successful if the size of the deobfuscated expression is smaller than or equal to the size of the ground truth, and the deobfuscated expression is semantically equivalent to the ground truth. Since the ground truth is not necessarily the smallest expression in terms of the semantics, a deobfuscated expression may be smaller than the ground truth.
- **Time**: We measure the time taken by the tools for deobfuscation with a timeout limit of 1 hour. In case of timeout, an intermediate result simplified so far is considered the final result in case of ProMBA and MBASolver. In case of Syntia, if the timeout occurs, the tool returns no result.

[5]In particular, NeuReduce can handle MBA expressions with up to 100 characters in length, which is not the case for most of the MBA expressions in our dataset.

5.2 Effectiveness of ProMBA

Deobfuscation Effectiveness. Table 2 summarizes the overall performance of ProMBA, MBASolver, and Syntia. Overall, ProMBA outperforms MBASolver in all metrics. ProMBA successfully simplifies 84.4% of the MBA expressions with an average time of 100 seconds. Most notably, the average size of the deobfuscated expressions produced by ProMBA is 9.39, which is smaller than the average size of the ground truth (9.64). On the other hand, MBASolver is able to successfully simplify only 13.2% of the MBA expressions with an average time of 141 seconds. MBASolver produces deobfuscated expressions with an average size of 344.5, which is much larger than the average size of the ground truth. Syntia simplifies only 39.4% of the MBA expressions with an average time 8.8 seconds. While Syntia is faster than ProMBA, it can only deobfuscate expressions whose ground truth is small (≤ 7 in AST size), which is reflected in the small average size of the deobfuscated expressions.

The performance of ProMBA is stable across the datasets. For every dataset, it generates smaller expressions than the ground truth on average. On the other hand, MBASolver generates larger expressions than the ground truth on average except for the MBA-Solver dataset, and Syntia often fails to deobfuscate MBA expressions in the MBA-Solver and QSynth datasets. This result suggests that ProMBA is not limited to a specific type of obfuscation, and generally applicable to the state-of-the-art MBA obfuscation techniques.

Table 2: Statistics of the deobfuscated expressions generated by ProMBA and MBASolver. The numbers in bold indicate the best results.

	Size (Avg.)			Success Rate			Time (Avg.)		
Dataset	**ProMBA**	**MBASolver**	**Syntia**	**ProMBA**	**MBASolver**	**Syntia**	**ProMBA**	**MBASolver**	**Syntia**
MBA-Solver	11.76	21.67	**4.61**	**80.31%**	25.16%	17.45%	65.64s	**6.3s**	12.95s
QSynth	17.48	77.71	**4.72**	**62.8%**	4.2%	22.8%	241s	64.83s	**12.37s**
Loki	3.51	866.25	**3.1**	**97.2%**	0.13%	74.4%	100.03s	347.77s	**2.07s**
Total	9.39	344.5	**3.55**	**84.44%**	13.19%	39.42%	100.36s	141.29s	**8.81s**

Table 3: Results for 15 randomly chosen deobfuscation tasks ("Obf." and "Orig.": the sizes of the obfuscated expression and the ground truth, "L.Time", "S.Time", "R.Time": the portion of time spent on the linear deobfuscator, synthesizer and term rewriter resp., "L.Reduce", "S.Reduce": the portion of size reduced by the linear deobfuscator and synthesizer resp., "# Rules": the number of rules, "# Rewrites": the number of rule applications, "Size": the size of the deobfuscated expression ("-" indicates a failure), "Time": the time taken for deobfuscation by each tool).

				MBASolver		Syntia		ProMBA								
Dataset	**ID**	**Obf.**	**Orig.**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**L.Time**	**S.Time**	**R.Time**	**L.Reduce**	**S.Reduce**	**#Rules**	**#Rewrites**
Loki	227	621	3	349	393s	3	0.15s	3	90s	42s	36s	12s	80.1%	19.9%	38	13
Loki	364	2438	3	1689	194s	3	0.29s	3	364s	33s	221s	110s	14.91%	85.09%	99	1274
Loki	1029	312	7	139	11s	-	0.14s	3	22s	15s	6s	1s	74.43%	25.57%	11	27
Loki	1124	753	6	208	29s	5	5.51s	5	30s	16s	11s	3s	44.92%	55.08%	14	61
Loki	1235	1139	7	950	88s	3	0.16s	2	21s	19s	1s	1s	11.52%	88.48%	5	19
MBA-Solver	410	53	17	25	0.04s	-	16.29s	15	136s	15s	120s	1s	50%	50%	13	25
MBA-Solver	857	91	22	27	0.04s	-	16.60s	21	249s	30s	217s	2s	78.57%	21.43%	13	34
MBA-Solver	1193	442	5	16	0.06s	-	1s	5	7s	3s	4s	0s	99.54%	0.46%	12	0
MBA-Solver	1350	260	6	10	0.05s	4	0.22s	4	6s	3s	2s	1s	80.08%	19.92%	13	1
MBA-Solver	1813	472	23	45	0.13s	-	15.26s	9	8s	3s	5s	0s	97.62%	2.38%	12	0
QSynth	211	929	19	249	145s	-	16.12s	7	90s	38s	40s	12s	15.62%	84.38%	23	44
QSynth	255	257	12	55	46s	-	16.08s	10	104s	21s	75s	8s	8.1%	91.9%	18	16
QSynth	283	62	10	24	49s	-	16.14s	10	66s	19s	46s	1s	15.38%	84.62%	15	2
QSynth	361	83	13	27	52s	-	15.61s	7	31s	8s	20s	3s	42.11%	57.89%	13	4
QSynth	465	395	17	115	54s	-	15.52s	43	854s	9s	820s	4s	0%	100%	16	88

Table 4: Statistics of the number of learned rules and application of the rules.

Dataset	**# of Rules (Avg.)**	**# of Applications (Avg.)**
MBA-Solver	12.7	9.05
QSynth	16.33	20.14
Loki	46.15	258.97
Total	25.66	103.9

Also, ProMBA is scalable enough to handle large expressions. With an average time of 100 seconds, it can simplify the Loki dataset comprising fairly large expressions (up to 34553 AST nodes). On the other hand, MBASolver is more than 3 times slower than ProMBA on average for the Loki dataset. Syntia can quickly deobfuscate the Loki dataset because the cost of sampling random input-output examples is irrelevant to the size of the expressions. However, the correctness of the deobfuscated expressions is not guaranteed.

Result in Detail. We discuss the results for each dataset in detail. Table 3 shows the results for 15 obfuscated expressions (5 for each dataset). Out of the 15 expressions, ProMBA generates smaller expressions than the ground truth in 14 expressions. In contrast, MBASolver cannot generate expressions as small as the ground truth in any of the expressions.

The results suggest the synergistic combination of linear MBA deobfuscation, synthesis, and term rewriting enables ProMBA's stable performance across all the datasets of different characteristics. For expressions containing many linear MBA sub-expressions (e.g., MBA-Solver #1193, #1682), the linear deobfuscator can quickly simplify a large portion of the expression, and the synthesis engine can simplify the remaining small portion of the non-linear sub-expressions, achieving a good efficiency. For expressions containing few linear sub-expressions (e.g., QSynth #465, Loki #1235), the synthesizer plays a more important role in simplifying the expression by synthesizing the non-linear sub-expressions, achieving a good coverage. For expressions for which specific complex MBA obfuscation rules are repeatedly applied (e.g., Loki #364), the term rewriter can efficiently simplify the expressions by reusing the learned rules, achieving a good efficiency.

In terms of time spent by linear MBA deobfuscation, synthesis, and term rewriting, synthesis is the most dominant in general. The linear MBA deobfuscation phase takes less than a minute in all

cases, and the term rewriting phase takes around 10 seconds except for one case (Loki #364). However, the synthesis phase takes up to 13 minutes. This is due to the large search space that the synthesizer has to explore and the time spent on SMT solving by the synthesizer. The synthesizer occasionally checks if a solution candidate is semantically equivalent to the original expression. This process is done by an SMT solver, which takes a significant amount of time when the original expression is large and complex. In case of QSynth #465, the original expression is a high-degree polynomial MBA expression for which checking equivalence is difficult for the SMT solver. The significant overhead of synthesis leads to the sub-optimal performance of ProMBA in terms of the size of the simplified expression because the synthesizer often fails to find the optimal solution within the time limit. The term rewriting phase usually takes less than 10 seconds except for Loki #364 where the term rewriting phase takes about 2 minutes. Because there are many rules that can be applied to the expression in Loki #364 (99 rules in total, which is the largest number of rules applied in all the deobfuscation tasks), and the target obfuscated expression is large (2438 nodes), searching for a sub-expression that can be replaced by another according to the rules takes a long time. Similar observations can be made for many other tasks in the Loki dataset. The number of rules learned in the Loki dataset is 46 on average, which is larger than the number of rules learned in the other datasets (16 and 12 on average for QSynth and MBA-Solver, respectively). Therefore, the term rewriting phase takes 4 and 16 times longer in the Loki dataset than in the QSynth and MBA-Solver datasets, respectively.

Learning Capability. We evaluate the learning capability of ProMBA by measuring the number of rewrite rules learned during the deobfuscation process and the number of times the learned rules are applied to rewrite expressions. Table 4 summarizes the results. On average, ProMBA learned an average of 25.7 rewrite rules and applied them 103.9 times. The rule sizes (the size of a rule $l \rightarrow r$ is measured by $|l|$) range from 3 to 224, with an average of 3.4. ProMBA learns fewer rules for the MBA-Solver dataset and applies them fewer times compared to the other datasets. This is because the MBA-Solver dataset contains many linear MBA sub-expressions which can be simplified by the linear MBA deobfuscator, leaving fewer opportunities for the synthesizers to learn rules.

We observe that ProMBA can learn and apply complex rules that can be hardly discovered by human experts. For example, the following intricate rules enable ProMBA to significantly simplify the expressions in the Loki dataset.

$$(\neg((y + y) \wedge x) \vee y) + ((y \vee (y \ll 1)) \wedge x) \rightarrow x \vee \neg y$$

On the other hand, MBASolver cannot simplify the left-hand side of the above rule because it is a non-polynomial MBA expression beyond the scope of MBASolver.

Failure Analysis. The inability of ProMBA to achieve success in 16% of the entire dataset is primarily attributed to the limited scalability of the SyGuS solver and the SMT solver. To investigate why ProMBA fails to deobfuscate some expressions, we randomly selected 100 failed cases evenly from the three datasets and attempted to double the allowed time for each synthesis attempt. As a result, 71 of the 100 cases became successful. However, the other 29 cases still failed even with the extended synthesis time due to the limitations of the SMT solver. In these instances, the SMT solver could not determine the equivalence between the ground truth and a solution candidate generated by the synthesizer within the time limit. This result suggests that as the scalability of the SyGuS and SMT solvers improves, the success rate of ProMBA will increase.

We also analyze the size of the deobfuscated expressions in the unsuccessful cases. We measure the ratio of the size of the deobfuscated expressions to the size of the ground truth. In the case of ProMBA, the average ratio in the MBA-Solver, QSynth, and Loki datasets is 1.22, 2.05, and 1.73, respectively. The average ratio for the entire dataset is 1.46. In the case of MBASolver, the average ratio in the three datasets is 1.81, 4.42, and 167.78, respectively. The average ratio for the entire dataset is 14.92. The result suggests that ProMBA can generate deobfuscated expressions that are closer to the ground truth across all datasets.

Summary of the Results. ProMBA can effectively deobfuscate a broad range of MBA expressions obfuscated by the state-of-the-art obfuscation tools, and is scalable enough to handle large expressions.

5.3 Comparison to the Baseline Tools

In addition to Table 2 that summarizes the overall performance of the three tools, Figure 3 visualizes the comparison among the tools in terms of the size of the deobfuscated expressions by each tool in each dataset. We sort the deobfuscated expressions by size and plot them in order of increasing size. The closer to the X-axis, the better the performance. In case of Syntia, the cases of timeout are not included in the plot because the tool does not generate any output for those cases.

Comparison to MBASolver. ProMBA significantly outperforms MBASolver in terms of all of the metrics as already shown in Table 2. In addition, according to Figure 3, ProMBA generates significantly smaller expressions compared to MBASolver in all datasets. In particular, for the Loki dataset, on average, ProMBA generates expressions 200 times smaller than MBASolver.

The major reason for the advantage of ProMBA over MBASolver is that the datasets include many MBA expressions that MBASolver cannot support. MBASolver gives up on simplifying a large portion of the expressions in the QSynth and Loki datasets because it cannot support many operators such as left shift, right shift, division, and modulo. Also, we observe the heuristic used by MBASolver often fails to simplify expressions that include non-linear sub-expressions. For example, for the expression $\neg(-((a \ll 2) \oplus ((a \ll 2))))$, MBASolver substitutes the sub-expression $(a \ll 2)$ with a new variable t hoping for generating a polynomial expression that can be simplified by its algebraic method. This results in the expression $\neg(-(t \oplus t))$, which is still non-polynomial and cannot be simplified by MBASolver. On the other hand, ProMBA does not have such limitations thanks to its synthesis-based approach.

Comparison to Syntia. ProMBA outperforms Syntia in terms of the success rate of deobfuscation as shown in Table 2, but Syntia

Table 5: Result of the comparison between three variants using different learning strategies.

	OnTheFly	*Offline*	*NoLearn*
Time (Avg.)	66.17	18.31	176.95
Size (Avg.)	8.34	255.29	8.93
Success Rate	91%	78%	88%

is faster than ProMBA. The average size of the deobfuscated expressions by Syntia is smaller than that of ProMBA, but it is due to the fact that Syntia can only generate small expressions as suggested by the distribution of the solution sizes of Syntia in Figure 3.

The reason for the efficiency of Syntia is that it does not check the equivalence between the original and deobfuscated expressions. It just checks the equivalence with respect to a limited number of inputs. On the other hand, ProMBA spends a significant amount of time to check the equivalence between original and deobfuscated sub-expressions for all inputs during the deobfuscation process. Because this equivalence check is expensive, ProMBA is slower than Syntia.

The cost for the better efficiency of Syntia is that it does not guarantee the correctness of the deobfuscated expressions. Syntia yields results that are not semantically equivalent to the original expressions for 11 deobfuscation tasks, whereas ProMBA always generates semantically equivalent expressions.

Summary of the Results. ProMBA outperforms the state-of-the-art baseline tools MBASolver in all aspects and Syntia in terms of the success rate of deobfuscation. Although Syntia is faster than ProMBA, it does not guarantee the correctness of the deobfuscated expressions, and it fails to generate a solution when the ground truth is large. ProMBA is not restricted to small ground truth expressions, and it guarantees the correctness of the deobfuscated expressions.

5.4 Efficacy of On-the-fly Learning of Rewrite Rules

We now evaluate the efficacy of the on-the-fly learning of rewrite rules. We compare the performance of three variants of ProMBA, each using a different learning strategy:

- *OnTheFly*: This variant learns and applies rewrite rules during the deobfuscation process (i.e., the original ProMBA).
- *Offline*: This variant is similar to [25]. It learns rewrite rules from a set of training deobfuscation tasks and applies them to target deobfuscation tasks. No new rules are learned and used in the target deobfuscation tasks.
- *NoLearn*: This variant relies only on sythesis for deobfuscation. No rules are learned for reuse from the synthesis attempts.

We compare the performance of three variants on 100 randomly chosen deobfuscation tasks from the entire datasets. The 100 tasks are evenly distributed across the datasets. For the *Offline* variant, we conduct 2-fold cross validation. That is, we use 50 randomly chosen tasks for training and the other remaining 50 tasks for testing and repeat the process with the two sets of tasks swapped to obtain the performance of the variant on the entire set of 100 tasks. Again, the training and testing tasks are evenly distributed across the datasets.

Table 5 summarizes the results. The *OnTheFly* variant outperforms the other two variants in terms of both the size of the deobfuscated expressions and the success rate. The *Offline* variant is the fastest. This is because it only applies rewrite rules without performing synthesis, which is the most time-consuming part. However, the average size of the deobfuscated expressions is the largest among the three variants, and the success rate is the lowest. That is because it often fails to deobfuscate expressions not covered by the rewrite rules learned from the training tasks. The *Offline* variant is even worse than *NoLearn* in terms of the success rate.

This result shows that relying on a pre-learned set of rewrite rules, which is the approach in [25], is not effective for MBA deobufscation due to the highly diverse rules used for MBA obfuscation.

The *NoLearn* is the slowest among the three variants. That is because it cannot reuse any rewrite rules and has to perform synthesis multiple times even for the same subexpressions. Therefore it often exhausts the time limit for each task and returns a suboptimal solution. This result shows that reusing learned rewrite rules is essential for efficient deobfuscation.

We further analyze the *Offline* variant to understand how sensitive it is to the training tasks. Instead of using the training tasks which are evenly distributed across the datasets, we use 50 tasks from the Loki dataset and the other 50 tasks from the other two datasets for training and testing and repeat the process with the two sets of tasks swapped. The average size of the deobfuscated expressions is 443.3, which is 1.7 times larger than the average size of 255.3 when the training tasks are evenly distributed across the datasets. This result shows that the *Offline* variant is sensitive to the training tasks and it is important to use a set of training tasks diverse enough to cover various obfuscation rules, which is difficult in practice since the obfuscation rules used in target obfuscated programs cannot be unknown in advance.

Overall, the results justify the need for our on-the-fly learning approach to deobfuscate MBA expressions efficiently and effectively.

Summary of the Results. The on-the-fly learning approach is essential for deobfuscating MBA expressions efficiently and effectively due to the highly diverse rules used in obfuscating MBA expressions. The offline learning approach [25] is highly sensitive to training data and exhibits a poor performance when the training data is biased.

5.5 Impact of the Underlying Synthesizer

We now evaluate the impact of the underlying synthesizer on the performance of ProMBA. We compare the performance of three variants of ProMBA, each using a different synthesizer: *w/SimbaS*, *w/Duet*, and *w/EUSolver* which use SimbaS, Duet [26], and EUSolver [4] as the underlying synthesizer, respectively. For synthesizing bit-vector expressions, SimbaS is the fastest, followed by Duet and EUSolver in terms of synthesis time [26, 34].

We compare the performance of three variants on the 100 deobfuscation tasks used in Table 5 in Section 5.4. Table 6 summarizes

Table 6: Result of the comparison between three variants using different synthesizers.

	w/SimbaS	*w/Duet*	*w/EUSolver*
Time (Avg.)	66.17	141.79	153.1
Size (Avg.)	8.34	10.38	10.69
Success Rate	91%	78%	76%
# of Rules (Avg.)	20.52	19.14	19.93
Size of Rules (Avg.)	18.05	17.99	11.15
# of Applications (Avg.)	95.9	83.34	79.22
Synthesis Time (Avg.)	52.06s	120.72s	136.92s

Table 7: Comparison of two variants of ProMBA with and without the optimizations in Section 3.6.

	w/Opt	*NoOpt*
Time (Avg.)	100.36	192.43
Size (Avg.)	9.39	28.51
Success Rate	84.44%	58.14%

the results. The *w/SimbaS* variant significantly outperforms the other two variants in all aspects. The time taken for synthesis by the *w/SimbaS* variant is more than two times shorter than the other two variants on average. The result shows that the *w/SimbaS* variant learns larger rules and applies them more frequently than the other two variants thanks to the best synthesis performance of SimbaS.

Summary of the Results. The performance of ProMBA is significantly affected by the performance of the underlying synthesizer.

5.6 Efficacy of the Optimizations

We now evaluate the effectiveness of the optimization techniques described in Section 3.6. For this purpose, we compare the performance of two variants of ProMBA over the entire benchmark set: *w/Opt* with all optimizations and *NoOpt* without any optimizations.

Table 7 summarizes the results. The *w/Opt* variant significantly outperforms the *NoOpt* variant in all aspects. When the optimizations are not applied, we can observe a 26% decrease in the success rate, a three-fold increase in the average size of the deobfuscated expressions, and a doubling of the average time taken for deobfuscation. We conclude that overall, the optimizations are significantly effective in improving the performance of ProMBA.

Summary of the Results. The optimizations described in Section 3.6 are essential for the effectiveness and efficiency of ProMBA.

6 DISCUSSION

We note the following opportunities for future improvements to our technique.

First, starting with initial rules that are hard to be proven by the SMT solver may address a potential attack on our approach. An adversary (e.g., a developer of an obfuscation tool) may subvert our approach by utilizing rules that are correct but challenging to be proven by SMT solvers. In our method, an SMT solver is employed to determine the equivalence between a solution candidate proposed by the synthesizer and the original obfuscated expression. However, SMT solvers may struggle to prove equivalence between two expressions if anyone involves certain complex (non-)polynomial MBAs or a large number of variables. For instance, Z3 is unable to prove the equivalence between $(x \wedge y) \times (x \vee y) + (x \wedge (\neg y)) \times ((\neg x) \wedge y)$ and $x \times y$ within an hour. If the adversary uses such rules for obfuscation, our technique may not be able to deobfuscate the expression due to the bottleneck of the SMT solver. To address this issue, we can start with initial rules including ones that are hard to be proven by the SMT solver.

Second, our technique can be improved if an underlying SyGuS solver can determine the unrealizability of SyGuS problems in the quantified bitvector domain (i.e., the absence of a solution in the search space). Our SyGuS problems encode the search for a smaller equivalent expression for a given expression. Because the existing SyGuS solvers are not able to determine the unrealizability of SyGuS problems in the quantified bitvector domain, they may spend a long time searching for a solution that does not exist. Meanwhile, the SyGuS solver may also spend a long time searching for a solution if the solution is complex. Therefore, users may be uncertain whether the cause of SyGuS solver's hang-ups is the complexity of the solution, or the unrealizability of the problem. If the SyGuS solver can efficiently determine the unrealizability of the problem, it will be able to quickly give up on the unrealizable problems and focus on the realizable ones, thereby improving the overall performance.

7 RELATED WORK

In this section, we discuss related work on term rewriting, program synthesis, and MBA deobfuscation.

Term Rewriting. Term rewriting has been broadly used in program transformation. Typical term rewriting systems depend on a set of rewrite rules manually designed by domain experts. The major drawback of this approach is that it is difficult to design a set of rewrite rules that can cover all possible transformations. This is also the case for MBA deobfuscation. SSPAM [18], a term rewriting-based MBA deobfuscation tool, uses a set of rewrite rules that are manually designed from various sources. Because of the incompleteness of the rewrite rules, SSPAM often fails to deobfuscate MBA expressions obfuscated by state-of-the-art obfuscators as shown in previous work [29, 39, 41]. To overcome this limitation, we propose a novel approach that automatically learns rewrite rules on the fly and applies them to deobfuscate MBA expressions. Similar to our approach, program synthesis has been used to learn rewrite rules for program optimization [6, 13, 24, 25, 37]. The major difference between our approach and the previous synthesis-based program optimization approaches is that we learn rewrite rules on the fly while those approaches learn them offline. As shown in our evaluation, our approach is more effective than the offline learning-based approach because MBA obfuscators often use a large number of rewrite rules all of which cannot be learned offline.

Program Synthesis. Over the the last decade, thanks to the standard SyGuS format [3] and the development of efficient solvers [7, 26, 42], program synthesis has been widely used in various applications. Deobfuscation of MBA expressions is also one of the appealing applications of program synthesis. Syntia [10] and Xyntia [31] use the Monte Carlo Tree Search (MCTS) algorithm and QSynth [16] uses an enumerative search strategy for deobfuscating MBA expressions. However, these approaches do not guarantee the correctness of deobfuscation results by synthesizing deobfuscated programs from a limited set of input-output examples. Our approach guarantees the correctness of deobfuscation results by synthesizing programs semantically equivalent to the given MBA expressions. In addition, our approach can leverage the state-of-the-art general-purpose program synthesis techniques by formulating the MBA deobfuscation problem as a SyGuS problem.

MBA Deobfuscation. As mentioned in Section 1, the previous approaches for MBA deobfuscation suffer from their own limitations. Arybo [22] transforms MBA expressions into bit-level expressions with only XOR and AND operators and then tries to simplify the bit-level expressions. However, this approach leads to a huge blowup in the size of the bit-level expressions, which makes it difficult to deobfuscate large MBA expressions. Neureduce [19] trains a neural network to deobfuscate MBA expressions. Given the vast number of possible MBA obfuscation rules, it is difficult to train a neural network that can cover all possible obfuscation rules. Also, the approach cannot guarantee the correctness of deobfuscation results. This is also the case for the previous synthesis-based approaches [10, 16, 31]. Algebraic simplification-based MBA deobfuscation techniques targeting linear MBA expressions have been proposed [29, 34]. These approaches are based on the algebraic property that if two linear MBA expressions are semantically equivalent for n-bit ($n \in \mathbb{N}$) input variables, they are also semantically equivalent for 1-bit input variables and vice versa. Given a linear MBA expression, they obtain a complete set of input-output behaviors of the expression by enumerating all possible combinations of zeros and ones for all 1-bit input variables and evaluating the expression for each combination. Then, a simpler equivalent expression from the obtained input-output examples is constructed. These approaches are quite effective for linear MBA expressions, and our method also leverages them for linear MBA expressions. However, they are not applicable to a broader class of MBA expressions.

Recently, Gamba, a more advanced algebraic simplification-based MBA deobfuscation tool has been proposed [35]. Gamba extends its predecessor SimbaD by introducing various hand-crafted algebraic simplification rules to deobfuscate non-linear MBA expressions. Although Gamba is efficient and effective, it is still limited to a restricted category of MBA expressions where the supported operators are limited to the logical operators $\wedge$, $\vee$, $\neg$, and $\oplus$ and the arithmetic operators $+$, $-$, $\times$ (and the left shift operator in a limited way). On the other hand, our approach can deobfuscate a much broader class of MBA expressions, handling all operators in the theory of fixed-width bit-vectors. In addition, relying on hand-crafted rules in Gamba makes it difficult to extend the tool to support a broader class of MBA expressions. As an example, an MBA expression $((2 \times a) - (a \wedge t)) - ((a - t) \wedge (a \vee t))$ cannot be deobfuscated by Gamba because it is beyond the scope of its simplifcation rules. On the contrary, our approach can deobfuscate it to $(a - t) \vee (a \vee t)$ thanks to the general-purpose program synthesis techniques.

8 CONCLUSION

In this paper, we presented a novel and versatile method for MBA deobfuscation called ProMBA, which overcomes the limitations of existing techniques by synergistically combining program synthesis, term rewriting, and algebraic simplification methods. Our method first simplifies linear MBA sub-expressions using an off-the-shelf deobfuscator, then recursively simplifies non-linear sub-expressions by synthesizing simpler sub-expressions and applying the resulting rewrite rules to other sub-expressions, until no further simplification is possible. We demonstrated the effectiveness of the approach on a large number of deobfuscation problems from various sources. The experimental results show that our method outperforms existing the state-of-the-art deobfuscation method.

ACKNOWLEDGMENTS

We thank the reviewers for insightful comments. The first author majors in Bio Artificial Intelligence. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944) and Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00995).

REFERENCES

[1] [n. d.]. Irdeto Cloaked CA: a secure, flexible and cost-effective CA system. https://irdeto.com/video-entertainment/conditional-access-system/.
[2] [n. d.]. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/.
[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD '13)*.
[4] 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*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336.
[5] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA.
[6] 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
[7] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-up Enumerative Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 227 (nov 2020), 29 pages. https://doi.org/10.1145/3428295
[8] Clark W. Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard Version 2.0.
[9] Fabrizio Biondi, Sebastien Josse, and Axel Legay. 2016. Bypassing Malware Obfuscation with Dynamic Synthesis. https://ercim-news.ercim.eu/en106/special/bypassing-malware-obfuscation-with-dynamic-synthesis.
[10] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko
[11] Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, Stefan Kowalewski and Anna Philippou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 174–177.
[12] Pierrick Brunet, Béatrice Creusillet, Adrien Guinet, and Juan Manuel Martinez. 2019. Epona and the Obfuscation Paradox: Transparent for Users and Developers,

a Pain for Reversers. In *Proceedings of the 3rd ACM Workshop on Software Protection* (London, United Kingdom) *(SPRO'19)*. Association for Computing Machinery, New York, NY, USA, 41–52. https://doi.org/10.1145/3338503.3357722

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

[14] Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. 2015. What's Decidable about Syntax-Guided Synthesis? arXiv:1510.08393 [cs.LO]

[15] C. S. Collberg, S. Martin, J. Myers, and B. Zimmerman. [n. d.]. Documentation for arithmetic encodings in tigress. http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic.

[16] Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. QSynth - A Program Synthesis based approach for Binary Code Deobfuscation. *Proceedings 2020 Workshop on Binary Analysis Research* (2020).

[17] Leonardo De Moura and Nikolaj Bjorner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[18] Ninon Eyrolles, Louis Goubin, and Marion Videau. 2016. Defeating MBA-Based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection* (Vienna, Austria) *(SPRO '16)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2995306.2995308

[19] Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. 2020. NeuReduce: Reducing Mixed Boolean-Arithmetic Expressions by Recurrent Neural Network. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 635–644. https://doi.org/10.18653/v1/2020.findings-emnlp.56

[20] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 519–531.

[21] Peter Garba and Matteo Favaro. 2019. SATURN - Software Deobfuscation Framework Based On LLVM. In *Proceedings of the 3rd ACM Workshop on Software Protection* (London, United Kingdom) *(SPRO'19)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3338503.3357721

[22] Adrien Guinet, Ninon Eyrolles, and Marion Videau. 2016. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *GreHack 2016 (Proceedings of GreHack 2016)*. Grenoble, France. https://hal.science/hal-01390528

[23] Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

[24] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[25] 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

[26] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

[27] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

[28] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. 2008. A Compiler-Based Infrastructure for Software-Protection. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (Tucson, AZ, USA) *(PLAS '08)*. Association for Computing Machinery, New York, NY, USA, 33–44. https://doi.org/10.1145/1375696.1375702

[29] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1701–1718. https://www.usenix.org/conference/usenixsecurity21/presentation/liu-binbin

[30] MapleSoft. [n. d.]. The Essential Tool for Mathematics. https://www.maplesoft.com/products/maple/.

[31] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. 2021. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2513–2525. https://doi.org/10.1145/3460120.3485250

[32] Camille Mougey and Francis Gabriel. 2014. DRM Obfuscation Versus Auxiliary Attacks. In REcon Conference.

[33] Ninon Eyrolles. 2017. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD. Université Paris Saclay. https://blog.quarkslab.com/resources/2017-06-09-nouthese-soutenance/thesis.pdf https://github.com/quarkslab/sspam/.

[34] Benjamin Reichenwallner and Peter Meerwald-Stadler. 2022. Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions. In *Proceedings of the 2022 ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks* (Los Angeles, CA, USA) *(Checkmate '22)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/3560831.3564256

[35] Benjamin Reichenwallner and Peter Meerwald-Stadler. 2023. Simplification of General Mixed Boolean-Arithmetic Expressions: GAMBA. https://arxiv.org/abs/2305.06763. In *Proceedings of the 2nd Workshop on Robust Malware Analysis, WORMA'23, co-located with the 8th IEEE European Symposium on Security and Privacy*. IEEE, Delft, The Netherlands.

[36] SageMath. [n. d.]. SageMath. http://www.sagemath.org/.

[37] 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

[38] 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)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

[39] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3055–3073. https://www.usenix.org/conference/usenixsecurity22/presentation/schloegel

[40] Wolfram. [n. d.]. Wolfram Mathematica. http://www.wolfram.com/mathematica/.

[41] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. 2021. Boosting SMT Solver Performance on Mixed-Bitwise-Arithmetic Expressions. 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, 651–664. https://doi.org/10.1145/3453483.3454068

[42] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1657–1681.

[43] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications*, Sehun Kim, Moti Yung, and Hyung-Woo Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–75.