MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation

Binbin Liu[*,2,1], Junfu Shen[1], Jiang Ming[3], Qilong Zheng[2], Jing Li[2], Dongpeng Xu[1]
[1] *University of New Hampshire*
[2] *University of Science and Technology of China*
[3] *University of Texas at Arlington*
binbin.liu@unh.edu, js1444@wildcats.unh.edu, jiang.ming@uta.edu, qlzheng@ustc.edu.cn, lj@ustc.edu.cn, dongpeng.xu@unh.edu

Abstract

Mixed Boolean-Arithmetic (MBA) obfuscation is a method to perform a semantics-preserving transformation from a simple expression to a representation that is hard to understand and analyze. More specifically, this obfuscation technique consists of the mixture usage of arithmetic operations (e.g., ADD and IMUL) and Boolean operations (e.g., AND, OR, and NOT). Binary code with MBA obfuscation can effectively hide the secret data/algorithm from both static and dynamic reverse engineering, including advanced analyses utilizing SMT solvers. Unfortunately, deobfuscation research against MBA is still in its infancy: state-of-the-art solutions such as pattern matching, bit-blasting, and program synthesis either suffer from severe performance penalties, are designed for specific MBA patterns, or generate too many false simplification results in practice.

In this paper, we first demystify the underlying mechanism of MBA obfuscation. Our in-depth study reveals a hidden two-way feature regarding MBA transformation between 1-bit and n-bit variables. We exploit this feature and propose a viable solution to efficiently deobfuscate code with MBA obfuscation. Our key insight is that MBA transformations behave in the same way on 1-bit and n-bit variables. We provide a mathematical proof to guarantee the correctness of this finding. We further develop a novel technique to simplify MBA expressions to a normal simple form by arithmetic reduction in 1-bit space. We have implemented this idea as an open-source prototype, named *MBA-Blast*, and evaluated it on a comprehensive dataset with about $10,000$ MBA expressions. We also tested our method in real-world, binary code deobfuscation scenarios, which demonstrate that MBA-Blast can assist human analysts to harness the full strength of SMT solvers. Compared with existing work, MBA-Blast is the most generic and efficient MBA deobfuscation technique; it has a solid theoretical underpinning, as well as, the highest success rate with negligible overhead.

[*]This work was done when Binbin Liu was a visiting scholar at the University of New Hampshire.

1 Introduction

Generally speaking, software obfuscation [1] is a transformation procedure to make a given program more difficult to analyze, while still preserving the program's original semantics. The competition between software obfuscation and deobfuscation schemes has evolved in the last years into an intensive arms race. On the one hand, many methods have been proposed in the literature to obfuscate software in different ways, generating a large body of literature on this topic. Amongst others, obfuscation techniques include encoding identifier names and data [2, 3], control flow flattening [4], opaque predicates [5, 6], run-time packers [7], and code virtualization [8, 9]. In practice, obfuscation techniques have been widely used in malicious software to hinder analysis [10, 11], digital right management (DRM) solutions [12, 13], and to protect secrets of cryptographic algorithms [14, 15]. As the rivals in this arms race, researchers have been working hard to understand or recover the original program behavior from the obfuscated form [16–23]. If there is any lesson we can learn from this body of work on improving deobfuscation techniques, it is that no single "silver bullet" can address all obfuscation schemes. One insight is that the status quo in software obfuscation development puts reverse engineers at a disadvantage: only having access to binary code greatly amplifies this asymmetry—the cost of deobfuscation is typically much higher than applying obfuscation.

In this paper, we focus on the analysis of an advanced obfuscation technique, called *Mixed Boolean-Arithmetic* (MBA) obfuscation [24]. MBA expressions are defined as the expressions that mix traditional arithmetic operators (e.g., $+, -, \times$) and Boolean operators (e.g., $\wedge, \vee, \neg, \oplus$). The effect of MBA obfuscation can transform a simple expression like $x+y$ to a complex, hard-to-understand expression with mixed Boolean and arithmetic operators, but the actual semantics of the new expression does not change. Existing math analysis theories only work either on pure Boolean expressions (e.g., normalization and constraint solving), or on pure arithmetic expressions (e.g., arithmetic reduction). So far, no publicly known

https://t.me/learningnets

methods, including both static and dynamic analysis-based methods, can effectively analyze or simplify MBA expressions. The root cause is that mixing two heterogeneous operators breaks regular reduction rules (e.g., the algebra laws of commutation, association, and distribution), which, in another word, ensures the practical strength of MBA obfuscation. Considering the distinct advantages in potency, resilience, and cost, MBA obfuscation has recently attracted the interests from security community: multiple research projects and industry products [9,25–30] have adopted this technique. Moreover, since many crypto algorithms also involve hybrid Boolean and arithmetic operations, MBA obfuscation has a broader impact on crypto analysis such as white-box cryptography [31,32].

The superior strength of MBA obfuscation has attracted research on software reverse engineering and deobfuscation. Existing publications have started working on simplifying MBA obfuscated expressions in an automated way, including bit-blasting [33], pattern matching [34], and program synthesis [21,35]. Unfortunately, state-of-the-art methods are still premature: they either can only analyze rather simple MBA expressions (due to the high performance cost), or they can only detect known MBA expressions in a range of fixed patterns. Many existing deobfuscation approaches only focus on the *syntactic* features of MBA expressions, but ignore the inner *semantics*. We feel the crux of these limitations is the lack of a deep understanding of MBA obfuscation mechanism, which has a solid mathematical foundation. In addition, no standard MBA expression benchmark exists to serve as a baseline for evaluating the effectiveness of an analysis method.

To bridge these gaps, we investigate the mathematical mechanism of MBA obfuscation and prove a hidden two-way transformation feature in the MBA obfuscation design: we discover that the MBA transformation behaves the same on 1-bit variables and any-length integers. Our finding reveals a new opportunity to directly simplify MBA expressions in 1-bit space. In light of this insight, we develop a novel technique, called *MBA-Blast*, to effectively reduce convoluted MBA expressions to simple forms. The key idea is to transform all bitwise expressions to specific MBA forms on 1-bit space and then perform arithmetic reduction. After replacing the bitwise operators, traditional arithmetic reduction laws can be smoothly applied, and they significantly promote the simplification efficiency. The correctness of our method is guaranteed by the two-way transformation feature, that is, the simplification result in 1-bit space is also correct in any-length integer space. We provide a mathematical proof to support this claim.

To demonstrate its practical viability, we implement MBA-Blast as an prototype and evaluate it on a comprehensive dataset including $10,000$ diversified MBA expressions. Our evaluation demonstrates that MBA-Blast significantly outperforms existing approaches. Only MBA-Blast succeeds in simplifying all obfuscated MBA expression with negligible overhead. We also evaluate MBA-Blast in assisting real-world obfuscated binary code analysis, such as solving MBA-powered opaque predicates with an SMT solver, analyzing virtualization obfuscated malware, and reverse engineering the encryption key generation algorithm used by a ransomware. Our results show that MBA-Blast is an appealing method to simplify MBA obfuscated expressions.

The impact of our work is mainly on areas related to software analysis. MBA-Blast can help human analysts simplify complexity expressions and understand their behaviors. From the view of arms race, our work also benefits the obfuscation community, because we expose the limitation of existing MBA design so that further improvements can be developed.

In summary, we make the following key contributions:

- We demystify the underlying mechanism of MBA obfuscation and identify a two-way transformation feature. The generated MBA rules have the same behavior on 1-bit Boolean variables and any-length integers. We are the first to prove the existence of this feature.
- This finding paves the way for our novel MBA deobfuscation technique, called *MBA-Blast*. Our method replaces bitwise operations with specific MBA expressions. In this way, we can seamlessly adopt arithmetic reduction rules to simplify MBA obfuscated expressions.
- Our proposed approach is implemented as a prototype evaluated on a comprehensive MBA benchmark and real-world environment. The result shows that MBA-Blast outperforms existing tools in terms of better accuracy and efficiency. MBA-Blast's source code and the MBA benchmark are available at `https://github.com/softsec-unh/MBA-Blast`.

2 Background

For pedagogical reasons, we first introduce the technical background needed to understand MBA obfuscation. Then we discuss the limitations of existing MBA deobfuscation work, which also serves as a motivation for our research.

2.1 MBA Expression

As noted above, Mixed-Boolean-Arithmetic (MBA) expressions mix Boolean operators ($\wedge, \vee, \neg, \oplus, \ldots$) and traditional integer arithmetic operations ($+, -, \times, \ldots$). Historically, MBA is known as smart tricks in algorithm optimizations. For instance, HAKMEM Memo [36] and Hacker's Delight [37] collect numerous identity equations involving addition and subtraction combined with logical operations. Two examples are listed as follows.

$$x - y = x + \neg y + 1 \tag{1}$$

$$x \oplus y = x \vee y - x \wedge y \tag{2}$$

https://t.me/learningnets

These two MBA identity equations are used for optimization purpose. Equation (1) shows how to build a subtracter from an adder and (2) presents a way to implement "exclusive or" using only three instructions, e.g., on a RISC machine. For a long time, MBA broadly scatters in various fields of computer science, e.g., optimization, data encoding, or compression, even without a formal name.

Zhou et al. [24, 38] extends the existing MBA concept to a more general model called "Boolean-arithmetic algebras", which generates MBA identities based on the following formal definition.

Definition 1. An MBA expression is:

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

where a_i is a constant coefficient, e_i are bitwise expressions of variables $x_1, \ldots, x_t$. $a_i e_i$ is called a term in the MBA expression.

Expression (3) gives a more complex MBA example within the definition above. The MBA includes 5 terms: $x, y, -x \wedge y, -3(x \oplus y)$ and 5. Note that if the Boolean expression is `True`, the term only has the coefficient, like the last term 5.

$$x + y - x \wedge y - 3(x \oplus y) + 5 \tag{3}$$

2.2 MBA Obfuscation

Because MBA identities expose the equivalence between two expressions, they are directly applicable to program obfuscation to transforms a simple expression into a complex form. For example, equation (1) and (2) can be used for obfuscating $x - y$ and $x \oplus y$. More similar MBA identity equations can be found in Hacker's Delight [37]. Eyrolles [39] and Banescu [40] enumerate a collection of MBA equations for obfuscation. Several MBA obfuscation rules for $x + y$ are listed as follows. Zhou et al. [24] prove that *any Boolean function has its non-trivial MBA expression equivalents*, which lays the theoretical foundation of MBA obfuscation.

$$\begin{aligned}
x + y &\rightarrow (x \vee y) + (\neg x \vee y) - (\neg x) \\
x + y &\rightarrow (x \vee y) + y - (\neg x \wedge y) \\
x + y &\rightarrow (x \oplus y) + 2y - 2(\neg x \wedge y) \\
x + y &\rightarrow y + (x \wedge \neg y) + (x \wedge y)
\end{aligned}$$

Due to the simplicity in implementation and the desirable mathematical principle, MBA obfuscation has captured interests widely from academia and industry. For example, Quarkslab [27], Cloakware [28], and Irdeto [29] include MBA obfuscation in their commercial products. Tigress [41], an academic C source code diversifier/obfuscator, encodes integer variables and expressions into complex MBA forms [25, 26]. Mougey and Gabriel [30] present a real-world MBA example found in an obfuscated Digital Rights Management (DRM) system. Blazy and Hutin [42] integrate formally verified MBA obfuscation rules into the generated binaries by the CompCert C compiler [43]. Recently, Xmark adopted MBA obfuscation to conceal the static signatures of software watermarking [13]. As malware authors always seek more advanced evasion techniques to stay under the detection radar, it did not take them long to become aware of the practical advantage of MBA obfuscation. ERCIM News reported in 2016 that MBA obfuscation has been detected in malware compilation chains [44]. We also observe MBA used in malware and virtualization obfuscation as shown in Section 7.6 and 7.7.

2.3 Strength of MBA Obfuscation

MBA obfuscation is ideally applicable for hiding sensitive variables and secret algorithms, such as magic numbers in cryptographic functions [45] and encryption key generation procedures in ransomware [46]. Compared to other obfuscation techniques, MBA obfuscation exhibits multiple distinct advantages. We elaborate on the strength of MBA obfuscation in terms of potency, resilience, cost, and correctness. The first three metrics were proposed by Collberg et al.'s pioneer work [47] to evaluate an obfuscation scheme. Correctness is another critical problem emerging from recent obfuscation development, but it has been largely overlooked in prior work.

Potency. Potency refers to how *complex* or *unreadable* the obfuscated result is to a human security analyst. MBA obfuscation places a heavy burden on human reverse-engineers in four ways: (1) significantly increases the number of Boolean and arithmetic operators; (2) introduces a multitude of new integers and bit-vectors; (3) hides the real parameters among them; (4) shuffles the calculation order. Manually reversing an MBA expression to its initial form is very challenging. Figure 1 shows an example of the code before and after MBA obfuscation.

MBA obfuscation impedes the effort of reverse engineering data structures from binary code [48]. A constant obfuscated by MBA can achieve the similar effect as an "opaque constant" [49]: it allows users to load a constant into a register, but static analyzers cannot determine the exact value. Like opaque constants, MBA obfuscation can be used to mislead the target of unconditional jump and call instructions, hide a variable's address, and complicate define-use chain analysis.

Resilience. Resilience represents the robustness of an obfuscation method in terms of resisting an automatic deobfuscator. Eyrolles [39] applies multiple simplification methods (e.g., mathematical reduction, compiler optimization, and SMT solver simplification) on expressions with MBA obfuscation, but none of them can effectively produce a correct simplification result. Bardin et al. present a novel technique in IEEE S&P'17 to assist obfuscated binary analysis, called *backward-bounded dynamic symbolic execution* [20]. However, the authors admitted that MBA obfuscation introduces

https://t.me/learningnets

```
int fun(int x,int y,int z)
{
  int c;
  c = x+y;




  return c;
}
```

(a) Original program.

```
int fun(int x,int y,int z)
{
  int c;
  c = 4*(~x&y)-(x^y)-(x|y)
      +4*~(x|y)-~(x^y)-~y-
      (x|~y)+1+6*x+5*~z+
      (~(x^z))-(x|z)-2*~x-
      4*(~(x|z))-4*(x&~z)
      +3*(~(x|~z));

  return c;
}
```

(b) MBA obfuscated program.

Figure 1: An example of MBA obfuscation for `x+y`, which is transformed into a complex expression mixing both arithmetic and boolean operations with a redundant variable `z`. A human analyst has a hard time to understand the new, obfuscated form.

hard-to-solve predicates, which hence become a major obstacle to their approach [50, 51]. In our evaluation, we use the state-of-the-art theorem solver, Z3 [52], to check the equivalence of the original expression and its MBA obfuscated form, but Z3 fails to return a result in five hours.

Cost. The cost of an obfuscation scheme includes two parts: instrumentation cost and run-time overhead. Instrumentation cost represents the time and resources for conducting the obfuscation transformation; run-time overhead refers to the slowdown and extra resource costs when the obfuscated program is running. MBA obfuscation adds very little overhead in terms of both types of cost. During obfuscation time, MBA transformations just rewrite the target expression with a new, complex but still equivalent MBA expression, without introducing any additional jump tables, function calls, or system calls. The obfuscation process can be directly applied to source code and easily combined with the normal compilation and linker workflow. The run-time overhead incurred by MBA obfuscation is also low because only simple boolean and arithmetic operations are involved. The new variables are directly located on the stack, so no extra cost comes from managing the heap memory blocks.

Correctness. Correctness means that the obfuscated program must behave exactly the same as the original program. Initially, correctness is easily guaranteed by designing individual obfuscation methods as a semantic-preserving transformation. However, as obfuscation methods are developed more and more complex, it becomes challenging to preserve program semantics after obfuscation. For instance, as one of the most sophisticated obfuscation techniques, code virtualization [8, 9] transforms part of a program to the bytecode in a new, custom virtual instruction set, and the bytecode is emulated by an embedded virtual machine at run time. Wang et al.'s study [53] points out that virtualization obfuscation results in program crash or incorrect output when 30% of the program is virtualized. Instead, MBA obfuscation is built on a solid mathematical basis, guaranteeing the correctness of obfuscation result. Recent work [42] has verified the correctness of a set of MBA obfuscation rules by using the formal proof system Coq [54].

2.4 Deobfuscation of MBA Expressions

On the other side of this arms race, researchers have explored the direction of reverse engineering and simplifying MBA expressions. Eyrolles' PhD thesis is the first work to go into this subject at full length [39]. Her experiments show that popular symbolic computation software such as Maple [55], Wolfram Mathematica [56], SageMath [57], and Z3 [52] fail to simplify MBA expressions because they do not support reduction rules for mixed bitwise and arithmetic operators. Furthermore, LLVM compiler optimizations [58] also have very limited effect on MBA simplification. Guinet et al. [33] present Arybo, a tool that normalizes MBA expressions to bit-level symbolic expressions with only $\oplus$ and $\wedge$ operations. However, the bloated size of bit-level expressions cause severe performance penalty, so Arybo can only deal with small-size MBA expressions. SSPAM [34] simplifies MBA expressions by a pattern matching algorithm. This method performs well on simplifying existing MBA examples and a real-world example [30]. As a common limitation of pattern matching techniques, it uses limited known rules to discover and reduce MBA expressions so it cannot handle generic MBA obfuscation. Biondi [35] presents an algebraic simplification to reduce the MBA complexity, but the method only works for specific MBA patterns, thus is also not generically effective. Blazytko et al. [21] leverage program synthesis techniques [59] to simplify MBA expressions by generating another simpler but equivalent expression. Due to the non-determinism and sampling mechanism of program synthesis, the correctness of simplification result is not always guaranteed.

The common limitation of existing deobfuscation efforts is that they treat MBA obfuscation as a black box, rather than investigate the mechanism under the hood. We also find that the lack of a standard and comprehensive MBA benchmark creates an obstacle: without a ground-truth benchmark, it is not clear how to compare these different methods.

3 How MBA Obfuscation Works: from One-bit to N-bit

In this section, we demystify the detailed underlying mechanism of MBA obfuscation. Zhou et al. [24] propose a systematic method to automatically generate MBA equations. By checking the truth table for t 1-bit variables, their method first seeks an MBA identity equation that holds for the t 1-bit variables, and then it deduces that the MBA equation also holds for any-length integer variables. In particular, for any $2^t \times k$ Boolean matrix with linearly dependent column vectors, it

https://t.me/learningnets

generates an MBA identity for t variables and k terms. The following example elaborates the procedure. Given a $2^2 \times 5$ Boolean matrix M ($t = 2$ and $k = 5$), we derive a bitwise expression for each column. The bitwise expressions involve two 1-bit variables, x and y. M essentially shows the truth table enumerating all possible values of x, y, and the bitwise expressions.

$$M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$
$$\quad x \quad y \quad x \vee y \quad \neg x \wedge y \quad x \wedge \neg y$$

Then we solve the linear equation system $M\vec{v} = 0$ and get the solution vector $\vec{v}$.

$$\vec{v} = \begin{pmatrix} 1 \\ 1 \\ -2 \\ 1 \\ 1 \end{pmatrix}$$

Regarding $\vec{v}$ as the coefficients, we produces an MBA identity as follows. The equation holds because the matrix M, treated as the truth table, exhaustively enumerates all possible values of the expressions.

$$x + y - 2(x \vee y) + (\neg x \wedge y) + (x \wedge \neg y) = 0$$

From this identity, an MBA obfuscation rule is easily constructed as follows:

$$x + y \rightarrow 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$$

Although so far this method only guarantees that the MBA identity holds for 1-bit variables, Zhou et al. [24] further prove it also holds for integers of any length. For simplicity, here we ignore the formal mathematical proof and give an imprecise description. For n-bit integers, every bit is treated separately when calculating the MBA expression. Because the identity holds for every bit, the whole calculation result also holds. Let X and Y be n-bit integers. $x_0, y_0, x_1, y_1, \dots$ represent every bit of the integer. The following calculation shows how the 1-bit identity is extended to an n-bit MBA expression:

$$X + Y - 2(X \vee Y) + (\neg X \wedge Y) + (X \wedge \neg Y) =$$

$$\sum \begin{cases} 2^0 \cdot (x_0 + y_0 - 2(x_0 \vee y_0) + (\neg x_0 \wedge y_0) + (x_0 \wedge \neg y_0)) \\ 2^1 \cdot (x_1 + y_1 - 2(x_1 \vee y_1) + (\neg x_1 \wedge y_1) + (x_1 \wedge \neg y_1)) \\ \dots \\ 2^{n-1} \cdot (x_{n-1} + y_{n-1} - 2(x_{n-1} \vee y_{n-1}) + (\neg x_{n-1} \wedge \\ \qquad y_{n-1}) + (x_{n-1} \wedge \neg y_{n-1})) \end{cases}$$

$$= 2^0 \cdot 0 + 2^1 \cdot 0 + \dots + 2^{n-1} \cdot 0$$
$$= 0$$

This method provides a systematic approach for constructing MBA equations. It is generic to cover simple cases such as shown in Hacker's Delight [37] and also the complex cases in Eyrolles [39] and Tigress [41].

4 Our Finding: "N-bit to One-bit" Also Holds

In this section, we present an exciting finding: the existing MBA obfuscation design actually implies a two-way transformation feature between 1-bit and n-bit variables. This finding paves the way for our deobfuscation method.

The approach in Section 3 successfully extends MBA identity from 1-bit space to integer space. Interestingly, the authors also vaguely mention that the reverse direction is also "plainly" correct. That means, if an MBA identity exists in integer space, then it must also hold in 1-bit space, which can be represented by the Boolean matrix described in § 3. However, the description provided by the authors was too brief to fully understand the proof procedure. It is not a trivial question because normal math reduction rules do not work by default within the context of MBA calculation. Eyrolles [39] also admitted that "we keep only one direction of the equivalence—this is the only direction we were able to prove, despite the other one being described as 'plain' by Zhou et al."

We wish to highlight that the correctness of n-bit to one-bit transformation is a matter of utmost importance: it will shatter the foundation of MBA obfuscation. If this proposition is proved as true, an integer MBA identity is the sufficient and necessary condition for the same form of MBA identity in 1-bit space. This implies that any integer MBA identity can be reduced to 1-bit space for simplification. Since 1-bit space is significantly smaller than integer space, the solution space for simplification and verification will be exponentially reduced, as we demonstrate later.

We prove the above proposition regarding n-bit to 1-bit transformation is true using *proof by contradiction*. To the best of our knowledge, we are the first to verify the correctness of this proposition. The detailed proof is shown as follows.

Definition 2. Let $E = \sum_{j=0}^{s-1} a_j e_j$ be an MBA expression, where a_j are integers and e_j are boolean functions $f_j(X_1, X_2, \dots, X_t)$ taking t variables $X_1, X_2, \dots, X_t$ as input. Each variable has n bits. We use $X_{k,i}$ to represent the ith bit of the kth input variable in e_j. Let M be the $2^t \times s$ boolean matrix representing the truth table of $e_0, e_1, \dots, e_{s-1}$. $\vec{v} = \begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{s-1} \end{pmatrix}$ is an s dimension vector consisting of all the coefficients in E.

Theorem 1. $E \equiv 0$ if and only if the linear system $M\vec{v} = 0$.

Proof. The sufficiency is proved in the MBA construction method [24], that is, if $M\vec{v} = 0$, then $E \equiv 0$. Now we prove

https://t.me/learningnets

the necessity, namely, if $E \equiv 0$, then $M\vec{v} = 0$.

If $E \equiv 0$, then

$$E = 2^0 \cdot E_0 + 2^1 \cdot E_1 + \ldots + 2^{n-1} \cdot E_{n-1} \equiv 0$$

where E_i is the calculation of E on the ith bit of input variables:

$$E_i = \sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \ldots, X_{t,i})$$

We prove $E_i = 0$ by contradiction.

Suppose $\exists k, E_k = \sum_{j=0}^{s-1} a_j f_j(X_{1,k}, \ldots, X_{t,k}) = \bar{e} \neq 0$. We construct a group of inputs $X'_1, X'_2, \ldots, X'_t$ where

$$\begin{cases} X'_{1,i} = X_{1,k} \\ X'_{2,i} = X_{2,k} \\ \ldots \\ X'_{t,i} = X_{t,k} \end{cases} \quad i = 1, 2, \ldots, n$$

Feed $X'_1, X'_2, \ldots, X'_t$ to E, then $\forall i = 1, 2, \ldots, n, E_i = \bar{e}$

$$\begin{aligned} E &= 2^0 \cdot E_0 + 2^1 \cdot E_1 + \ldots + 2^{n-1} \cdot E_{n-1} \\ &= 2^0 \cdot \bar{e} + 2^1 \cdot \bar{e} + \ldots + 2^{n-1} \cdot \bar{e} \\ &= (2^n - 1)\bar{e} \end{aligned}$$

Because $E \equiv 0$,

$$(2^n - 1)\bar{e} = 0$$

$$\bar{e} = 0$$

This contradicts the supposition that $\bar{e} \neq 0$. Hence, our supposition is false, so for any input $X_{1,i}, X_{2,i}, \ldots, X_{t,i}$,

$$E_i = \sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \ldots, X_{t,i}) = 0$$

$$a_0 e_0 + a_1 e_1 + \ldots + a_{s-1} e_{s-1} = 0$$

Therefore,

$$M\vec{v} = 0$$

□

Essentially our proof shows that, if an n-bit MBA identity

$$E(X_1, X_2, \ldots, X_t) \equiv 0$$

holds, the same identity also holds in one-bit space

$$E(x_1, x_2, \ldots, x_t) \equiv 0$$

This conclusion completes the " two-way transformation" feature in MBA obfuscation, which sheds a light on our new approach to reversing MBA obfuscation.

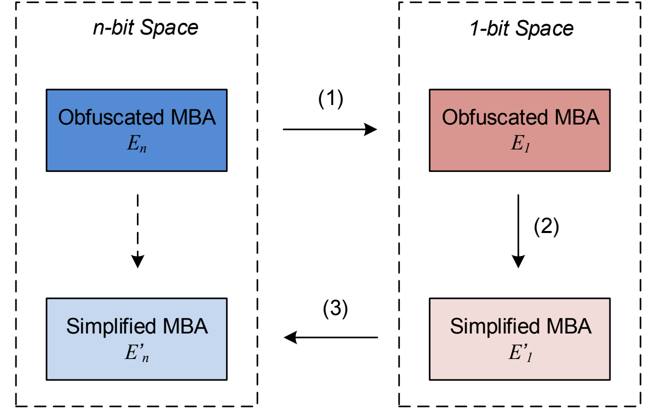


Figure 2: The logic flow of MBA-Blast simplification. (1) Transform the Obfuscated MBA expression from n-bit to 1-bit space. (2) Simplify the MBA in 1-bit space. (3) Transform the simplified MBA from 1-bit to n-bit space.

5 MBA-Blast

The "two-way" feature in current MBA obfuscation implies that any n-bit obfuscated MBA expression can be simplified in 1-bit space. Consequently, the MBA reduction in 1-bit space is equivalent to that in n-bit space. This idea enlightens us to design a novel method, called *MBA-Blast*, to simplify n-bit MBA expression.

5.1 Approach

Our key idea is to develop MBA simplification rules in 1-bit space and use them to simplify any n-bit complex MBA expression. Figure 2 shows the logic flow. Given an n-bit obfuscated MBA expression E_n, our goal is to find a simple and equivalent n-bit expression E'_n as the simplified result (as indicated by the dashed arrow). Theoretically, our simplification includes three steps as follows.

(1) Transform E_n in n-bit space to E_1 in 1-bit space.
(2) Find a simplified MBA expression E'_1 in 1-bit space, such that $E_1 - E'_1 \equiv 0$.
(3) Transform E'_1 in 1-bit space to E'_n in n-bit space.

Step (1) and (3) have been proved in Section 3 and 4, which means, any n-bit MBA identity is equivalent to the same form on 1-bit space.

$$E_n - E'_n \equiv 0 \quad \Leftrightarrow \quad E_1 - E'_1 \equiv 0$$

Therefore, the simplification problem boils down to Step (2): finding a simple MBA form E'_1 to satisfy the 1-bit MBA identity $E_1 - E'_1 \equiv 0$.

The unique benefit of reducing the problem to 1-bit space is that, *we can use truth tables to enumerate all possible values*. 1-bit variables only have two possible values, 0 and 1, so it

https://t.me/learningnets

Table 1: Truth table of $x \vee y$ and $x \wedge y$.

x	y	$x \vee y$	$x \wedge y$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

largely reduces the searching space when investigating MBA equation. Taking the truth table in Table 1 as an example, applying the method in Section 3 generates the following 1-bit MBA equation,

$$x + y - (x \vee y) - (x \wedge y) = 0$$

which means,

$$x \vee y = x + y - (x \wedge y)$$

In this way, we can build MBA equations for two-variable truth table ($2^4 = 16$ different cases), as shown in Table 2. For ease of presentation, the first column presents the truth values as a 4-digit binary string, e.g., the truth value of $x \vee y$ is 0111. Note that the truth value 1111 is represented as -1 to guarantee MBA equations are valid on a ring [24].

The interesting finding in Table 2 is that, all the 16 cases can be represented as a linear combination of x, y, $x \wedge y$, and -1. In other words, any two-variable 1-bit expression can be transformed to an MBA expression with the following general form, where c_1, c_2, c_3, c_4 are coefficients.

$$c_1 x + c_2 y + c_3 (x \wedge y) - c_4$$

This finding forms the foundation of our simplification method. According to Definition 1, an MBA expression $\sum a_i e_i$ is essentially a linear combination of 1-bit expressions. After replacing all 1-bit expressions with the corresponding MBA forms in Table 2 and combining like terms, the original MBA expression will be reduced to a simple form including only 4 terms: x, y, $x \wedge y$, and a constant.

$$\begin{aligned}\sum a_i e_i &= \sum a_i (c_{1i} x + c_{2i} y + c_{3i} (x \wedge y) - c_{4i}) \\ &= C_1 x + C_2 y + C_3 (x \wedge y) - C_4\end{aligned}$$

The following example shows how to simplify the obfuscated MBA expression in Section 3, $x + y \to 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$.

$$\begin{aligned}&2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) \\ &= 2(x + y - x \wedge y) - (y - x \wedge y) - (x - x \wedge y) \\ &= 2x + 2y - 2(x \wedge y) - y + (x \wedge y) - x + (x \wedge y) \\ &= x + y\end{aligned}$$

This procedure produces an MBA identity equation $2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) = x + y$ in 1-bit space. According to the "two-way" feature, this equation also holds in n-bit space.

Table 2: Enumeration of all the Bool-Arithmetic rules used in MBA-Blast.

Truth Value	Boolean Expr	MBA Expr
0000	0	0
0001	$x \wedge y$	$x \wedge y$
0010	$x \wedge \neg y$	$x - (x \wedge y)$
0011	x	x
0100	$\neg x \wedge y$	$y - (x \wedge y)$
0101	y	y
0110	$x \oplus y$	$x + y - 2 * (x \wedge y)$
0111	$x \vee y$	$x + y - (x \wedge y)$
1000	$\neg(x \vee y)$	$-x - y + (x \wedge y) - 1$
1001	$\neg(x \oplus y)$	$-x - y + 2 * (x \wedge y) - 1$
1010	$\neg y$	$-y - 1$
1011	$x \vee \neg y$	$-y + (x \wedge y) - 1$
1100	$\neg x$	$-x - 1$
1101	$\neg x \vee y$	$-x + (x \wedge y) - 1$
1110	$\neg(x \wedge y)$	$-(x \wedge y) - 1$
1111	-1	-1

Therefore, $x + y$ is the simplification result of $2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$ in n-bit space. A more complex example is shown in Appendix C, which simplifies the MBA expression in Figure 1.

The distinct advantage of our method is that it guarantees to simplify an MBA expression to a normal simple form, with only low-cost arithmetic computation. The "two-way" feature guarantees the simplification result seamlessly working in 1-bit space and n-bit space.

5.2 MBA-Blast Algorithm

The method above is able to simplify one MBA expression. However, in practice, a complex MBA expression may include multiple sub-expressions obfuscated by different MBA equations. We need to apply the simplification to each sub-expression recursively until no sub-expression can be simplified any more. The whole procedure is described as Algorithm 1.

The algorithm takes an expression E as input and returns its simplified form. First, it traverses all sub-expressions of E and marks it as reducible if the sub-expression is an MBA. Then, for each reducible sub-expression e, the algorithm first replaces every bitwise operation with the MBA expression in Table 2 (ReplaceBoolWithMBA) and then performs conventional arithmetic reduction (ArithReduce) to get the normal form. Next, the ReplaceMBAWithBool function tries to match the normal MBA form with the simple bitwise expression in Table 2, e.g., $-y + (x \wedge y) - 1$ is replaced by $x \wedge \neg y$. If e' is simpler than e, which means the simplification is successful, e' is used for updating the whole expression E. Otherwise, e' is already the simplest form, so the algorithm marks e as

https://t.me/learningnets

Algorithm 1 MBA-Blast Algorithm

```
1: Input: MBA expression E
2: function MBA-BLAST(E)
3:     for e_s ∈ SubExpr(E) is MBA do
4:         e_s ← reducible
5:     end for
6:     while e ∈ SubExpr(E) is reducible do
7:         ReplaceBoolWithMBA(e)
8:         e' ← ArithReduce(e)
9:         ReplaceMBAWithBool(e')
10:        if e' is simpler than e then
11:            update(E, e')
12:        else
13:            e ← irreducible
14:        end if
15:    end while
16:    return E
17: end function
```

irreducible and continues to work on other reducible sub-expressions. The complexity of e' and e are measured by the number of their Directed Acyclic Graph (DAG) nodes. More detailed discussion about complexity measurement of MBA expressions is presented in Section 7. The algorithm keeps simplifying MBA sub-expressions and it terminates when no reducible sub-expression is available.

6 Implementation

We implement the algorithm as an analysis prototype called MBA-Blast. Figure 3 shows an overview of MBA-Blast's architecture and how it interacts with other analysis tools. The prototype accepts inputs from various front-ends, simplifies MBA expressions, and outputs the results in different formats. In total, the whole implementation includes a front-end interface, the main MBA-Blast program, and a back-end interface. The front-end interface receives MBA expressions from different sources (e.g., an execution trace, the disassembled code from IDA Pro [60], or source code) and translates the code to an intermediate representation (IR) for MBA-Blast to process. MBA-Blast simplification consists of four major components. First, a parser reads the obfuscated formula and builds the Abstract Syntax Tree (AST). Second, a tree substitution component substitutes bitwise operations with specific MBA expressions. After that, each AST is translated to IR by a formula generation step. The last component applies arithmetic reduction laws to the formulas and outputs the simplified results to the back-end interface. The back-end interface can translate the IR to different outputs, for example, human-readable formulas or SMT-LIB code for theorem provers such as Z3. MBA-Blast is designed as a tool that can easily work with binary analysis tools, solvers, and compilers.

The whole prototype is written in 2800 lines of Python code. The parser, AST substitution, and formula generation components are developed based on Python `AST` library. We leverage the `SymPy` library for arithmetic simplification and solving linear equation systems. We design an representation for efficiently analyzing, transforming, and interpreting MBA symbolic formulas. We also develop several utilities for measuring the quantitative metrics of MBA expressions, such as counting the number of DAG nodes and MBA alternations.

7 Evaluation

In this section, we conduct a set of experiments to evaluate MBA-Blast. We have four objectives in mind: *correctness*, *effectiveness*, *practicability*, and *performance*. In particular, we design experiments to answer the following four research questions (RQs).

1. **RQ1:** Is the simplified result equivalent to the original MBA expression? *(correctness)*
2. **RQ2:** Compared to the original complicated MBA expression, how much complexity is reduced by MBA-Blast? *(effectiveness)*
3. **RQ3:** Is MBA-Blast able to assist security experts in real-world software reverse engineering? *(practicability)*
4. **RQ4:** How much overhead does MBA-Blast introduce? *(performance)*

As the answer to RQ1, we apply MBA-Blast to simplify two MBA datasets where the ground truth (correctly simplified form) is available. We use Z3 solver [52] to check whether every simplified result is equivalent to the ground truth. For RQ2, we calculate and compare the complexity metrics such as number of DAG nodes and number of MBA alternation. We also run Z3 on the original and simplified MBA to compare the solving time. For RQ3, we perform case studies to show MBA-Blast's practicability, including analyzing the output of an MBA obfuscator, solving MBA-powered opaque predicates, and reverse-engineering virtualized malware and a ransomware sample. In response to RQ4, we study MBA-Blast's performance data such as running time and memory footprint.

7.1 Experimental Setup

Datasets. We aim to evaluate MBA-Blast using a large number of diverse MBA expressions. First, we checked existing resources including MBA expressions [24, 30, 36–39] and collected 62 MBA obfuscation equations as the first dataset. The number of existing MBA examples is quite deficient for a systematic study. We also find these examples are biased as well. For instance, they only include a limited diversity of bitwise expression patterns like $x \wedge y$ and $x \vee y$.

We notice that new MBA identity equations can be simply extended from the linear combination of existing MBA

https://t.me/learningnets

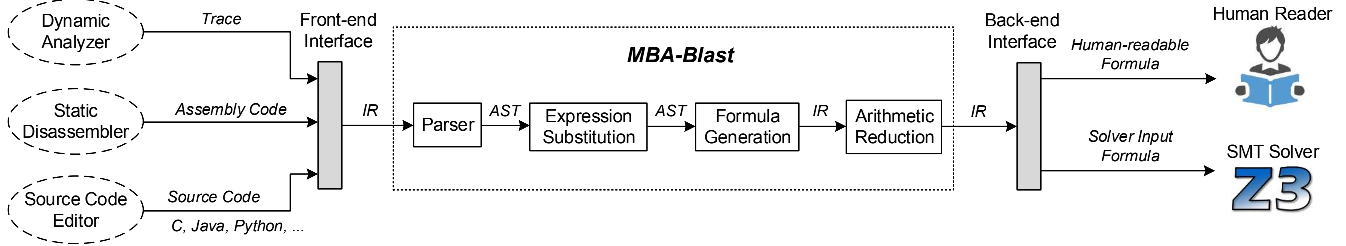


Figure 3: An overview of MBA-Blast's workflow. The words in italics represent the format between two components.

identities. Figure 4a shows an example. By multiplying -2 to the second MBA identity and then adding to the first one, it extends a new MBA obfuscation expression. Furthermore, this extension can also produce multiple-variable MBA expressions as shown in Figure 4b.

$$x \oplus y = y + x - 2(x \wedge y)$$
$$x \wedge y = (x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$$
$$\Downarrow$$
$$x \oplus y = y + x - 2(x \vee y) + 2(\neg x \wedge y) + 2(x \wedge \neg y)$$

(a) Generate new MBA identity by linear combination.

$$x + y = 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$$
$$x \wedge z = -(x \oplus z) + z + (x \wedge \neg z)$$
$$\Downarrow$$
$$x + y + z = 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y) + (x \wedge z) + (x \oplus z) - (x \wedge \neg z)$$

(b) Generate multiple-variable MBA identity.

Figure 4: Extend MBA identities by linear combination.

These extensions synthesize the Dataset 2 including $10,000$ MBA expressions. Every sample in the dataset is a 3-tuple: $\langle C, S, M \rangle$. C is the complex MBA form, S is the simple form, and M records the meta data. Note that, S is the correct simplified result, i.e., ground truth, for every complex MBA C. To guarantee the diversity of the dataset, we control the following features, which are calculated and saved as the meta data M in every sample.

- **Length of variables.** The dataset covers different variable length, including 8 bits, 16 bits, 32 bits, and 64 bits. Each category contains $2,500$ MBA expressions.
- **Number of variables.** The number of input variables ranges from 1 to 10.
- **Number of terms.** Since new MBA expressions can be generated by linear combination, the number of terms is another feature for controlling complexity. Number of terms in this dataset ranges from 3 to 80.

Peer Tools for Comparison. We collect existing, state-of-the-art MBA deobfuscation tools and run them on the same datasets as the comparison baselines. The latest version of three open source tools are downloaded from GitHub for comparison: Arybo [33], SSPAM [34], and Syntia [21]. Arybo is a Python tool for transforming MBA formulas to a bit-level symbolic representation. SSPAM (Symbolic Simplification with PAttern Matching) is a tool for simplification of MBA expressions written in Python. It uses `SymPy` for arithmetic simplification, and Z3 for flexibly matching equivalent expressions with different representations. Syntia is a program synthesis framework for synthesizing obfuscated code's semantics. It produces input-output pairs from instruction traces and then synthesizes a code snippet's semantic based on these input-output pairs.

Machine Configuration. All of our experiments are running on a testbed machine with Intel Xeon W-2123 4-Core 3.60GHz CPU, 64GB 2666MHz DDR4 RAM, 2.5TB SSD Hard Drive, Running Ubuntu 18.04 OS.

7.2 Dataset 1: Collected MBA Examples

In the first experiment, we run MBA-Blast and other peer tools on Dataset 1, which contains all MBA expressions collected from existing works. The simplification result is evaluated from two aspects, *correctness* and *effectiveness*.

Correctness means the expressions before and after simplification must be semantically equivalent. We use Z3 solver [52] to perform equivalence checking. The challenge here is most of the MBA expressions before simplification are too complex for Z3 to solve. Since we have the initially un-obfuscated expressions as the ground truth, our alternative is to check equivalence between the simplified result and the ground truth. Note that even a correctly simplified result may have different syntax with its ground truth (see the example in Table 4), so the equivalence checking step is indispensable.

The other aspect, *effectiveness*, reflects how much complexity is reduced by the simplification method, so we measure and compare the expression complexity before and after simplification. Eyrolles [39] introduces three metrics to measure MBA complexity: number of nodes, MBA alternation, and average bit-vector size. Because SSPAM, Syntia, and MBA-Blast do not change bit-vector size and Arybo always reduces

https://t.me/learningnets

Table 3: Comparative evaluation results using Dataset 1. In "# of Correctness" column, "Yes" means equivalent, "No" means not equivalent, and "T.O." means time out (Z3 fails to return a result in five hours), and "Ratio" indicates the ratio of outputs passing equivalence checking. "Average # of Nodes", "Average # of MBA Alternation", and "Average Processing Time" report the result on *correctly* simplified results. "Before" represents obfuscated MBA expressions to be simplified, and "After" represents the simplified expressions delivered by different deobfuscation tools. "Average Processing Time" reports the average time that each tool takes to process one MBA sample.

Method	# of Correctness				Average # of Nodes			Average # of MBA Alternation			Average Processing Time
	Yes	No	T.O	Ratio (%)	Before	After	A/B (%)	Before	After	A/B (%)	(Seconds/Sample)
Arybo	37	0	25	59.7	9.1	27.3	300.0	2.1	0.0	0.0	30.2
SSPAM	62	0	0	100.0	9.4	7.9	84.0	2.3	1.5	65.2	4.6
Syntia	59	3	0	95.2	9.4	4.6	48.9	2.3	0.5	26.1	8.9
MBA-Blast	62	0	0	100.0	9.4	4.7	50.0	2.3	0.5	21.7	0.009

Table 4: Correct simplification result appears different, but it is semantically equivalent to the ground truth.

Ground Truth	Before	After
$-3(x\wedge\neg y)$	$4(\neg x\wedge y)-(x\oplus y)+3\neg(x\vee y)+\neg(x\oplus y)-\neg y-\neg x-(\neg x\vee y)-\neg(x\wedge y)$	$3(x\wedge y)-3x$

any length variable to 1-bit variable, so measuring bit-vector size is trivial in this experiment. We use the rest two quantitative metrics to measure MBA complexity.

1. **Number of DAG nodes.** An MBA expression is translated to a Directed Acyclic Graph (DAG), where the nodes are operators, variables, and constants. The number of nodes in the DAG is a metric for describing the expression complexity.
2. **MBA Alternation.** A key source of MBA complexity comes from mixing integer arithmetic operations and bitwise operations. We adopt "MBA alternation" to measure the number of operations that connect different types of operations. For example, in $x\wedge y+2z$, the $+$ represents an MBA alternation, because its left operand is a bit-vector generated by $x\wedge y$, and its right operand is an integer arithmetic $2z$.

For these complexity metrics, a larger value indicates a more complex MBA expression. We expect the metrics' values will decrease after simplification. Table 3 shows the evaluation result on Dataset 1. For this and the following experiment, we set five hours as a practical timeout threshold for Z3 solving.

Compared to the existing tools, all MBA-Blast's outputs pass correctness testing, and their complexity measurement values are considerably reduced. We observe that Arybo performs well on simple MBA expressions. However, when handling complex expressions, Arybo's simplification result is even more complex than the original expression. For over 1/3 (25 out of 62) of the samples, it generates very complex formulas that cannot be solved by Z3 within the time threshold. The reason is that Arybo breaks all integers to 1-bit variables causing the result size to increase drastically. The simplification result from Arybo does not have MBA alternation, because it reduces all arithmetic operators to bitwise operators. SSPAM successfully simplifies majority of the samples as they are included in SSPAM's pattern matching library. The core technique of Syntia is stochastic program synthesis, which approximates program semantics using Monte Carlo Tree Search (MCTS). Syntia's simplification largely relies on the quality of sampling input-output pairs. When the sampling points *perfectly* represent the MBA expression, it can achieve a correct, simplified form, and its complexity reduction is on a par with MBA-Blast. Because the samples in Dataset 1 are not very complex, Syntia can correctly synthesize majority of them (59/62). The last column shows MBA-Blast only introduces negligible processing overhead compare to the peer tools.

7.3 Dataset 2: Comprehensive MBA Dataset

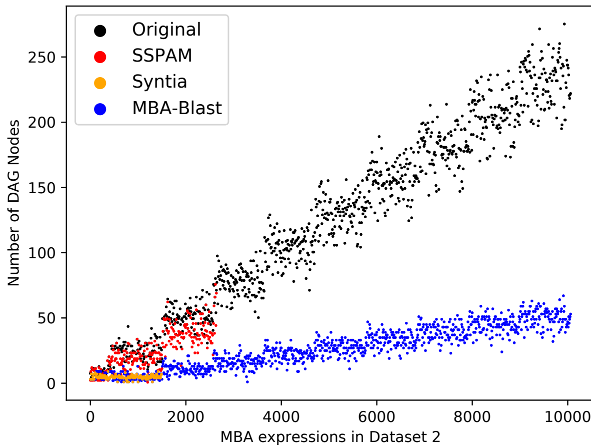
As the second experiment, we run MBA-Blast and other baseline tools on Dataset 2. The result in Table 5 presents an obvious gap between other tools and MBA-Blast}. Only MBA-Blast successfully generates verifiable simplification results for all MBA samples. The average processing time for each case is less than 0.1 second, significantly faster than existing tools.

Because the MBA samples in Dataset 2 are diverse and well-labeled, this experiment reveals more detailed findings. Arybo can handle 431 MBA samples in Dataset 2, all of which are small-size, 8-bit MBA (average number of DAG nodes is 13.4). For the rest of cases, Arybo generates size-explosion results (over $20,000$ DAG nodes) that exceed Z3 solver's processing capacity. SSPAM can process more complex MBA samples using its pattern library, and the average number of DAG nodes is 32.1. For other MBA samples, SSPAM either returns an incorrect result or crashes with a segmentation error. Syntia can output a simplified expression for every MBA sample in Dataset 2, but up to 85.6% of them are not correct result due to the imprecise "guess" in program synthesis. On

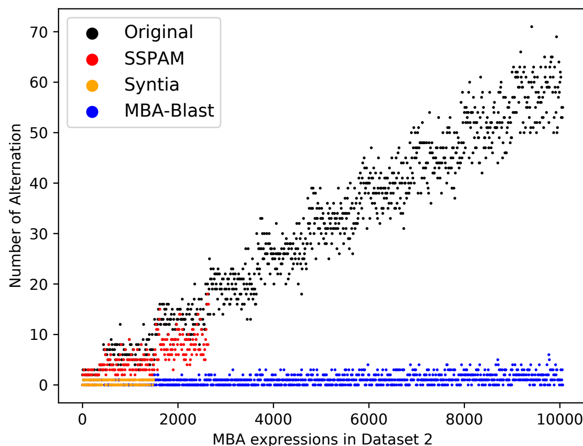
https://t.me/learningnets

Table 5: Comparative evaluation results using Dataset 2. In "# of Correctness" column, "Yes" means equivalent, "No" means not equivalent, and "T.O." means time out (Z3 fails to return a result in five hours), and "Ratio" indicates the ratio of outputs passing equivalence checking. "Average # of Nodes", "Average # of MBA Alternation", and "Average Processing Time" report the results on *correctly* simplified results. "Before" represents obfuscated MBA expressions to be simplified, and "After" represents the simplified expressions delivered by different deobfuscation tools. "Average Processing Time" reports the average time that each tool takes to process one MBA sample.

Method	# of Correctness				Average # of Nodes			Average # of MBA Alternation			Average Processing Time
	Yes	No	T.O	Ratio (%)	Before	After	A/B (%)	Before	After	A/B (%)	(Seconds/Sample)
Arybo	431	0	9,569	4.3	13.4	25.5	190.3	3.4	0.0	0.0	640.7
SSPAM	2,550	0	7,450	25.5	32.1	25.1	78.3	8.5	5.4	63.5	438.2
Syntia	1,438	8,562	0	14.4	26.4	4.6	4.0	6.4	0.5	1.6	9.3
MBA-Blast	10,000	0	0	100.0	113.2	19.5	17.2	30.8	0.9	2.9	0.053



(a) Number of DAG Nodes.



(b) Number of MBA Alternation.

Figure 5: The distribution of two complexity metrics on Dataset 2. We compare the MBA expression before simplification with the simplified results from SSPAM, Syntia, and MBA-Blast. We do not plot Arybo's results because they increase the complexity metrics' values.

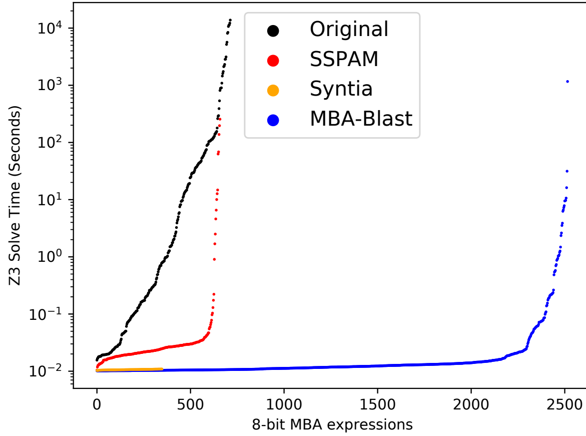
the samples that Syntia synthesizes the correct result, its performance rivals MBA-Blast, as shown in Figure 5a and 5b. The figures zoom in two complexity metrics (Number of DAG nodes and MBA alternation) and plot the distribution. Syntia's dots are very close to MBA-Blast's dots, indicating that they compete with each other in terms of complexity reduction; however, MBA-Blast can correctly simplify considerably more MBA expressions.

Moreover, Figure 6 presents Z3's solving time when performing correctness testing for different length variables in Dataset 2. For simplicity, we only present 8-bit and 64-bit graphs and the complete result is shown in Appendix B. The curve density represents how many expressions are verified as correct. The black curve represents MBA samples before simplification. When the length of variable increases from 8-bit to 64-bit, the density of black curve becomes sparser. It is because when variable length growths, the searching space becomes larger and Z3 has more difficulty to solve the formula. Compared to that, the density of blue curve, representing MBA-Blast's result, does not change and it covers all samples in the Dataset 2. That means, considerable MBA samples were not solvable before MBA-Blast's simplification, but they can be solved very quickly after MBA-Blast's simplification. Before simplification, only $1,542$ MBA expressions in Dataset 2 pass correctness testing. After MBA-Blast's simplification, Z3 can solve 6.5X more expressions.

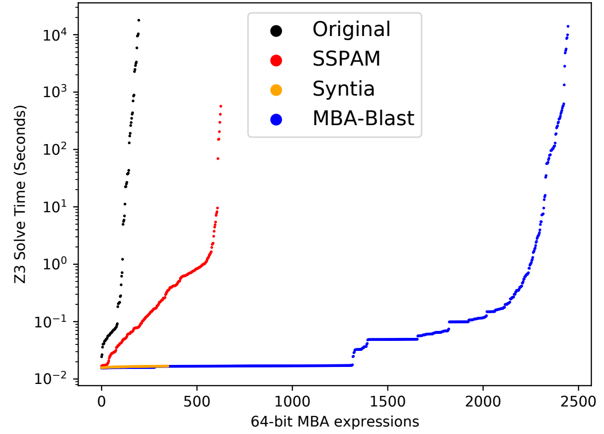
The trend and slope of these curves represent the change of expression complexity before and after simplification. For those variables with short length, the majority of blue curve has a small slope, which means most of the simplified result can be solved quickly. As the length of variables increase from 8-bit to 16-bit, more part of the curve has a large slope. That means, due to the increasing search space, Z3 spends more time to verify the simplification result when the variable length is large. The simplification results from SSPAM are more complex than MBA-Blast and Syntia's results are competent, but both of them fail to generate correct results for majority of samples in the dataset.

In addition, we observe that the number of input variables also affects Z3's verification time. Overall, the solving time increases with the number of inputs. For MBA samples involving more than 8 input variables, Z3 spends considerable

https://t.me/learningnets



(a) 8-bit result.



(b) 64-bit result.

Figure 6: Z3 solving time when handling different data length in Dataset 2.

time to verify it, although MBA-Blast correctly generates the simplification result.

$$x+y = x \vee y + x \wedge y$$
$$x-y = x \wedge \neg y - \neg x \wedge y$$

(a) MBA obfuscation rules in Tigress.

$$\begin{aligned} x-y+z &= (x-y)+z \\ &= ((x-y)\vee z) + ((x-y)\wedge z) \\ &= ((x\wedge\neg y - \neg x \wedge y)\vee z) + \\ &\quad ((x\wedge\neg y - \neg x\wedge y)\wedge z) \end{aligned}$$

(b) Tigress recursively generates a complex MBA expression.

$$\begin{aligned} &((x\wedge\neg y-\neg x\wedge y)\vee z)+((x\wedge\neg y-\neg x\wedge y)\wedge z) \\ &= ((x-x\wedge y-y+x\wedge y)\vee z)+((x-x\wedge y \\ &\qquad -y+x\wedge y)\wedge z) \\ &= ((x-y)\vee z)+((x-y)\wedge z) \\ &= (t\vee z)+(t\wedge z) \qquad (\mathbf{x-y\rightarrow t}) \\ &= (t+z-t\wedge z)+t\wedge z \\ &= t+z \\ &= x-y+z \qquad (\mathbf{t\rightarrow x-y}) \end{aligned}$$

(c) MBA-Blast simplification steps.

Figure 7: Tigress's complex MBA expression and MBA-Blast's simplification.

7.4 Defeating Tigress MBA Obfuscation

We are interested in applying MBA-Blast in real-world obfuscation scenario to check its *practicability*. Tigress [41] is an automated software obfuscation tool with MBA obfuscation embedded. We first randomly generate 1,000 C functions as the testbed and then use the `EncodeArithmetic` option in Tigress to obfuscate these functions.

One interesting observation is that Tigress can recursively apply MBA obfuscation transformation to generate complex result. For example, Figure 7a shows two obfuscation transformation in Tigress. By recursively applying these rules, it translates $x-y+z$ to a more complex MBA expression in Figure 7b.

Our evaluation result shows that MBA-Blast successfully simplifies all of the obfuscated output from Tigress, including these complex cases. As shown in Algorithm 1, MBA-Blast keeps simplifying sub-linear MBA expressions, so the whole obfuscated expression is simplified in a bottom-up way. Figure 7c shows how MBA-Blast simplifies a complex MBA expression generated by Tigress.

7.5 Solving MBA-Powered Opaque Predicates

Opaque predicate is a prevalent software obfuscation technique to complicate control flow. This method has been widely adopted by obfuscation tools such as Obfuscator-LLVM [61]. Recently, deobfuscation methods based on symbolic execution [20, 62] and machine learning [63] have been proposed to detect and reverse engineer opaque predicates in programs. However, opaque predicates can be further protected by MBA obfuscation to hide the static features and generate more variants. MBA-powered opaque predicates bring new challenges to symbolic execution and machine learning based countermeasures. First, both Backward-bounded DSE [20] and LOOP [62] rely on SMT solvers to check whether a predicate is opaquely true or false, but as we have shown, SMT solvers cannot solve complex MBA expressions in a

https://t.me/learningnets

reasonable time. Second, a large number of heterogeneous MBA obfuscation rules can be created by the method used in Dataset 2, so it is very hard for machine learning methods [63] to learn the patterns of MBA obfuscation.

In this experiment, we show that MBA-Blast can assist solving MBA-powered opaque predicates. The simplified output of MBA-Blast removes the complexity of MBA obfuscation, hence it unleashes the power of other opaque predicate reverse tools. We collect commonly used opaque predicates from existing work [20, 62] as follows.

$$\begin{aligned}
\forall x \in Z. \quad & x^2 + x \bmod 2 \equiv 0 \\
\forall x \in Z. \quad & x(x+1)(x+2) \bmod 3 \equiv 0 \\
\forall x, y \in Z. \quad & 7y^2 - 1 \neq x^2 \\
\forall x \in Z. \quad & (x^2+1) \bmod 7 \neq 0 \\
\forall x \in Z. \quad & (x^2+x+7) \bmod 81 \neq 0 \\
\forall x \in Z. \quad & (4x^2+4) \bmod 19 \neq 0 \\
\forall x \in Z. \quad & x^2(x+1)^2 \bmod 4 \equiv 0
\end{aligned}$$

We apply MBA obfuscation to these predicates and create 70 variants. For example, we apply $x+y \rightarrow 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$ to $x^2 + x \bmod 2 \equiv 0$ and the new opaque predicate is:

$$\forall x \in Z. \quad 2(x^2 \vee x) - (\neg x^2 \wedge x) - (x^2 \wedge \neg x) \bmod 2 \equiv 0$$

In our experiment, we use Z3 to solve the 70 MBA-powered opaque predicates, but Z3 does not return any result in the time limit of five hours. In contrast, MBA-Blast successfully simplifies all MBA-powered opaque predicates. Then we apply Z3 to the outputs of MBA-Blast, and it solves the results within the similar time as that in previous work. Therefore, this experiment demonstrates that the simplification result from MBA-Blast helps to harness the full strength of SMT solver-based deobfuscation methods.

7.6 MBA Usage in Real-World Malware

An interesting question is the popularity of MBA obfuscation in real-world malware. Unfortunately, unlike binary packers that reveal distinctive features (e.g., entropy deviation and code-to-data ratio [23]), blindly searching the presence of MBA obfuscation in malware binaries is a nontrivial task—being stealthy is another advantage of MBA obfuscation. Even so, we observe MBA integrated into commercial software obfuscator VMProtect [9]. In this experiment, we study the usage of MBA in the malware obfuscated by VMProtect.

VMProtect is one of the most sophisticated obfuscators that are also widely used in malware. For example, in May 2019, hackers infected over 50,000 servers around the world with cryptocurrency mining malware, whose kernel-mode rootkit is protected by VMProtect to frustrate reverse engineers and malware researchers [64, 65]. VMProtect translates program code into custom bytecode and interprets the bytecode at run time via an embedded emulator, so that the original code never reappears in memory. In addition, VMProtect applies MBA obfuscation to further complicate the operations in bytecode handlers.

Table 6: MBA in malware obfuscated by VMProtect. "N" is the number of malware samples in each category. "# with MBA" shows the number of samples that include MBA. "MBA Expr" reports the number of MBA expressions detected from the samples in each category. Avg. # of Nodes and Avg. MBA alternation reports the average MBA complexity in each category.

Category	**N**	**Size (MB)**		**# with MBA**	**MBA Expr**	**Avg. # of Nodes**	**Avg. MBA Alternation**
	(132)	min	max	(105)	(157)		
Trojan	36	0.2	12.5	30	41	6.7	1.6
Virus	33	0.1	15.9	26	40	7.5	1.7
Malware	33	0.1	15.3	26	41	5.3	1.3
Riskware	8	0.6	24.4	7	11	9.8	2.6
CoinMiner	7	2.4	9.5	6	9	10.9	2.9
Backdoor	4	0.4	6.7	2	3	8.0	2.0
ADware	4	0.2	6.6	3	4	8.8	2.0
Rsmware	3	0.7	9.3	3	5	10.6	3.0
Spyware	2	0.3	10.5	1	2	10.0	3.0
Others	2	0.3	0.7	1	1	5.0	2.0

To investigate the usage of MBA in VMProtect obfuscated malware, we collect 132 samples from VirusTotal [66] by searching the keywords "vmprotect" and "vmp". To guarantee the collected samples are up-to-date, we restricted the "Last Submission Date" from 2020/05 to 2020/09. For every sample, we first identify the fetch-dispatching cycle in the virtual machine. Next, we extract the handlers that performs various VM operations, such as addition and subtraction. By manually inspecting the behaviors of these handlers, we find that MBA are used in VMProtect handlers for encoding arithmetic and bitwise computation. For example, VMProtect uses the following MBA to encode subtraction:

$$x - y = \neg(\neg x + y) \wedge \neg(\neg x + y)$$

Table 6 summarizes the collected VMProtect malware samples and the detected MBA with complexity metrics. Among the 132 malware samples, we identify 157 MBA expressions in 105 samples from different categories. It indicates that MBA widely exist in diverse types of VMProtect malware, from traditional Trojan and virus to modern ransomware and spyware. Appendix A shows more complex MBA samples collected from these malware.

For all the MBA expressions identified from malware samples, MBA-Blast successfully simplifies them to a concise, human-readable form. For example, the following procedure shows how MBA-Blast simplifies the obfuscated expression above on the right side of the equation and produces the result $x - y$. We also run Z3 to verify the correctness of the simplification result. This experiment shows that MBA-Blast

https://t.me/learningnets

can effectively simplify the MBA in virtualization obfuscated malware so that malware analysts are released from tedious manual reverse engineering.

$$
\begin{aligned}
&\neg(\neg x+y)\wedge\neg(\neg x+y) \\
&=\neg t\wedge\neg t && (\mathbf{\neg x+y\rightarrow t}) \\
&=-t-t+t\wedge t-1 \\
&=-t-1 \\
&=-(\neg x+y)-1 && (\mathbf{t\rightarrow \neg x+y}) \\
&=-(-x-1)-y-1 \\
&=x-y
\end{aligned}
$$

7.7 Case Study: Ransomware Analysis

This section presents our experience of using MBA-Blast to analyze a ransomware sample[1] collected from VirusTotal. Since MBA obfuscation can transform bitwise operations to complex forms with trivial runtime overhead, it is well suited for obfuscating crypto algorithms. Ransomware is an infamous malware type that intensively relies on crypto algorithms to encrypt the victims' files. We run the collected ransomware sample in a sandbox and set up Intel Pin [67] to record the ransomware execution. After that, we investigate the recorded trace and identify suspicious MBA transformations. This ransomware encrypts users' files using AES-256 algorithm and shreds files before removing them from disk. In the record trace, we observe a suspicious MBA behavior happening before entering the AES algorithm, so we doubt that the MBA transformation is related to key generation or initialization vector (IV). By carefully reverse engineering the binary code, we understand the ransomware's behavior and confirm that the malware developer adopts MBA to obfuscate both the encryption key and IV. More specifically, the ransomware generates a key and an IV for every file it encrypts, and then it appends the key and IV to the end of file after encryption. After the victim pays ransom to the malware developer, a decryption process is invoked to extract the key and IV from every file for decryption. Malware authors must protect the key and IV, otherwise victims can obtain them and then recover their files by running AES-256 decryption algorithm without paying ransom. We observe an MBA expression taking the encryption key K and a constant C as inputs, and then use MBA-Blast to simplify it. The MBA and MBA-Blast's simplification procedure is listed as follows. The final simplification result is $K\oplus C$. Therefore, the malware developer hides the encryption key by calculating $\oplus$ with a magic number C. MBA obfuscation is adopted for protecting the $\oplus$ operation. Similarly, we discover that the IV is also protected by $\oplus$ with a different constant. This case study shows that, although MBA-Blast is not particularly designed as a malware analysis tool, it can help understand behaviors of obfuscated malware.

[1] MD5: `218ee40649267be13d85c6ff0a91b603`

$$
\begin{aligned}
&(K\vee\neg C)+(\neg K\vee C)-2*(\neg(K\vee C))-2*(K\wedge C) \\
&=-C+K\wedge C-1+(-K+K\wedge C-1)- \\
&\qquad 2(-K-C+K\wedge C-1)-2(K\wedge C) \\
&=K+C-2(K\wedge C) \\
&=K\oplus C
\end{aligned}
$$

7.8 Performance

This section shows MBA-Blast's performance data. Table 7 presents the time and memory cost when MBA-Blast processes MBA expressions with different complexity level. MBA-Blast is very effective because it does not rely on any search or heuristic method. Our implementation is based on `AST` and `SymPy` Python library, which can perform expression substitution and arithmetic reduction efficiently. Overall, MBA-Blast only introduces a negligible overhead.

Table 7: MBA-Blast's performance on MBA expressions with different complexity.

# of Nodes	Time (Second)	Memory (MB)
10	0.0128	0.2
100	0.0528	0.5
200	0.0964	0.6
300	0.1358	0.7

8 Discussion

MBA-Blast demonstrates the feasibility and scalability of automatically reducing complex MBA expressions. However, we also note some potential opportunities for future improvement as below.

First, the simplification result from MBA-Blast may not be the simplest form. The normal output form is $c_1x+c_2y+c_3(x\wedge y)-c_4$. While it does significantly reduce the MBA complexity, this form is not guaranteed as the simplest result. Table 4 has provided one example. The ReplaceMBAWithBool function alleviates this problem by reversely applying the transformation in Table 2. MBA-Blast can be further extended to mitigate this problem by adding more rules in Table 2 so that it can produce more diverse simplification result.

Similarly, an adversary may attack MBA-Blast by intentionally applying multiple rounds of MBA substitution. The correct simplification requires precisely understanding the dependency between these rounds. Current MBA-Blast implementation only handles simple dependence cases, i.e., substituting all common sub-expressions. A more precise dependency analysis will be helpful to address this limitation.

It is also possible that attackers combine MBA obfuscation with other data encoding techniques to create complex expressions with bitwise and arithmetic operations but does not meet

https://t.me/learningnets

the MBA definition in this paper. MBA-Blast is designed for resolving MBA expressions, so unfortunately it does not have the capacity to directly reverse other obfuscation methods. It is interesting to further investigate whether MBA-Blast can benefit other de-obfuscation techniques.

9 Conclusion

This paper tackles a data obfuscation scheme, Mixed Boolean-Arithmetic (MBA) obfuscation, which uses both bitwise and arithmetic operations to generate an unintelligible expression. The cost of applying MBA obfuscation is rather low, but the resulting expression becomes a tough challenge for reverse engineering attempts, including advanced binary code analysis utilizing SMT solvers. The existing efforts to counter MBA obfuscation either work in an ad-hoc manner or suffer from heavy overhead. In this paper, we investigate the underlying mechanism of MBA obfuscation and prove a hidden two-way transformation feature between 1-bit and n-bit variables. This finding enlightens us to develop MBA-Blast, a novel MBA deobfuscation technique. The key idea is to simplify MBA expressions to normal forms and then perform arithmetic reduction in 1-bit space. Our large-scale MBA deobfuscation experiment and real-world malware study demonstrate MBA-Blast's efficacy and generality. Developing MBA-Blast not only advances automated software reverse engineering, but also delivers a benchmark serving as a baseline for future research in this direction.

Acknowledgments

We would like to thank our shepherd Lorenzo Cavallaro and the anonymous paper and artifact reviewers for their helpful feedback. We especially thank Thorsten Holz for the insightful suggestions. We also thank VirusTotal for providing the academic API and malware samples. This research was supported by NSF grant CNS-1948489. Jiang Ming was supported by NSF grant CNS-1850434.

References

[1] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Computing Surveys*, 49(1), April 2016.

[2] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, chapter 4.4, pages 258–276. Addison-Wesley Professional, 2009.

[3] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, chapter 13, pages 269–296. No Starch Press, 2012.

[4] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of Software-Based Survivability Mechanisms. In *Proceedings of International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, 1998.

[6] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method. In *Proceedings of the 19th Information Security Conference (ISC'16)*, 2016.

[7] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[8] Oreans Technologies. Code Virtualizer: Total Obfuscation against Reverse Engineering. http://oreans.com/codevirtualizer.php, 2019.

[9] VMProtect Software. VMProtect software protection. http://vmpsoft.com, 2019.

[10] Kevin A. Roundy and Barton P. Miller. Binary-code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 46(1), 2013.

[11] Philip OKane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The Hidden Malware. *IEEE Security and Privacy*, 9(5), 2011.

[12] Christian Collberg and Clark Thomborson. Software Watermarking: Models and Dynamic Embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, 1999.

[13] Haoyu Ma, Chunfu Jia, Shijia Li, Wantong Zheng, and Dinghao Wu. Xmark: Dynamic Software Watermarking Using Collatz Conjecture. *IEEE Transactions on Information Forensics and Security*, 14(11), March 2019.

[14] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-Box Cryptography and an AES Implementation. In *International Workshop on Selected Areas in Cryptography*, 2002.

[15] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. A White-Box DES Implementation for DRM Applications. In *ACM Workshop on Digital Rights Management*, 2002.

[16] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[17] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[18] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[19] Babak Yadegari and Saumya Debray. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[20] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

[21] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the Semantics of Obfuscated Code. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[22] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partial-Virtualized Binary Code Simplification. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

https://t.me/learningnets

[23] Binlin Cheng, Jiang Ming, Jianming Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[24] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Proceedings of the 8th International Conference on Information Security Applications (WISA'07)*, 2007.

[25] Christian Collberg, Sam Martin, Jonathan Myers, and Bill Zimmerman. Documentation for Arithmetic Encodings in Tigress. http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic.

[26] Christian Collberg, Sam Martin, Jonathan Myers, and Bill Zimmerman. Documentation for Data Encodings in Tigress. http://tigress.cs.arizona.edu/transformPage/docs/encodeData.

[27] Quarkslab. Epona Application Protection v1.5. https://epona.quarkslab.com, July 2019.

[28] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. A Compiler-based Infrastructure for Software-protection. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'08)*, 2008.

[29] Irdeto. Irdeto Cloaked CA: a secure, flexible and cost-effective conditional access system. www.irdeto.com, 2017.

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

[31] Hamilton E. Link and William D. Neumann. Clarifying Obfuscation: Improving the Security of White-Box DES. In *International Conference on Information Technology: Coding and Computing*, 2005.

[32] Andrey Bogdanov and Takanori Isobe. White-Box Cryptography Revisited: Space-Hard Ciphers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[33] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *Proceedings of GreHack 2016*, 2016.

[34] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO'16)*, 2016.

[35] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of Synthesis in Concolic Deobfuscation. *Computers & Security*, 70, 2017.

[36] Michael Beeler, R William Gosper, and Richard Schroeppel. Hakmem. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1972.

[37] H.S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.

[38] Yongxin Zhou and Alec Main. Diversity via Code Transformations: A Solution for NGNA Renewable Security. *The National Cable and Telecommunications Association Show*, 2006.

[39] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. PhD thesis, Université Paris-Saclay, 2017.

[40] Sebastian Banescu and Alexander Pretschner. Chapter Five - A Tutorial on Software Obfuscation. Advances in Computers. 2018.

[41] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed Application Tamper Detection via Continuous Software Updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, 2012.

[42] Sandrine Blazy and Rémi Hutin. Formal Verification of a Program Obfuscation Based on Mixed Boolean-arithmetic Expressions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'19)*, 2019.

[43] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7), July 2009.

[44] Fabrizio Biondi, Sébastien Josse, and Axel Legay. Bypassing Malware Obfuscation with Dynamic Synthesis. https://ercim-news.ercim.eu/en106/special/bypassing-malware-obfuscation-with-dynamic-synthesis, July 2016.

[45] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

[46] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. PayBreak: Defense Against Cryptographic Ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*, 2017.

[47] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, The University of Auckland, 1997.

[48] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[49] Andreas Moser Christopher Kruegel and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[50] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-oriented Protections). In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*, 2019.

[51] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. Obfuscation: Where Are We in anti-DSE Protections? (a First Attempt). In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'19)*, 2019.

[52] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, 2008.

[53] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. Translingual Obfuscation. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P'16)*, 2016.

[54] The Coq development team. The Coq proof assistant reference manual Version 8.9.1. http://coq.inria.fr, 2019.

[55] MapleSoft. The Essential Tool for Mathematics. https://www.maplesoft.com/products/maple/, 2020.

[56] WOLFRAM. WOLFRAM MATHEMATICA. http://www.wolfram.com/mathematica/, 2020.

[57] sagemath. SageMath. http://www.sagemath.org/, 2020.

[58] Peter Garba and Matteo Favaro. SATURN – Software Deobfuscation Framework Based on LLVM. In *Proceedings of the 3rd International Workshop on Software Protection (SPRO'19)*, 2019.

[59] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[60] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011.

https://t.me/learningnets

[61] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM–Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*, 2015.

[62] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[63] Ramtine Tofighi-Shirazi, Philippe Elbaz-Vincent, Irina Mariuca Asavoae, and Thanh-Ha Le. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. In *Proceedings of the 3rd International Workshop on Software Protection (SPRO'19)*, 2019.

[64] Lindsey O'Donnell. 50k Servers Infected with Cryptomining Malware in Nansh0u Campaign. http://tiny.cc/vj9zsz, May 2019.

[65] Ed Targett. Chinese Hackers Dropped Rootkit in 50,000 Servers: Then Left Theirs Wide Open. https://www.cbronline.com/news/guardicore-chinese-hackers-servers, May 2019.

[66] VirusTotal. VirusTotal Intelligence: Combine Google and Facebook and apply it to the field of Malware. https://www.virustotal.com/gui/intelligence-overview, 2020.

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

A MBA Samples from VMProtect malware

We list more complex MBA samples found in VMProtect malware as follows.

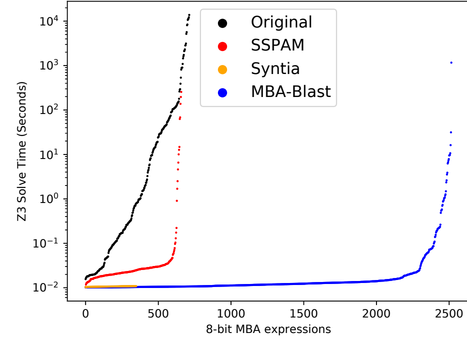
```
-x-y = ~(~(~(-1+x)|~(-1+x))+y)&~(~(~(-1+x)|~(-1+x))+y)
     = ~(~(~(-1+x)&~(-1+x))+y)&~(~(~(-1+x)&~(-1+x))+y)

x+1-y = ~(~(x+1)+y)|~(~(x+1)+y)
      = ~(~(x+1)+y)&~(~(x+1)+y)

a+b = ~(~(a+b)|~(a+b))|~(~(a+b)|~(a+b))

-x+~y =  (~(-1+x)|~(-1+x))+(~y&~y)

~a+x&y = (~a&~a)+(~(~x|~y)|~(~x|~y))

x&y+a&b = (~(~x|~y)|~(~x|~y)) + (~(~a|~b)|~(~a|~b))
        = (~(~x&~x)&~(~y&~y)) + (~(~a&~a)&~(~b&~b))

x|y+a|b = (~(~x&~y)&~(~x&~y)) + (~(~a&~b)&~(~a&~b))
        = (~(~x|~x)|~(~y|~y)) + (~(~a|~a)|~(~b|~b))
```

B Z3 Solving Time Comparison on Dataset 2

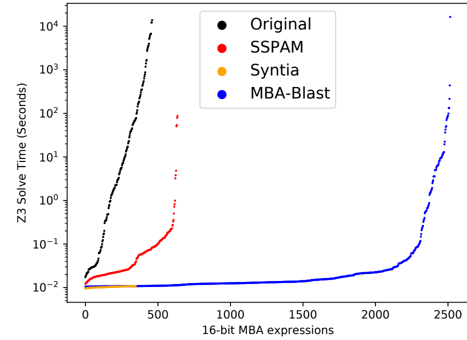
Figure 8 presents the complete simplification result of 8-bit, 16-bit, 32-bit, and 64-bit MBA samples in Dataset 2.

C A Complex MBA Example

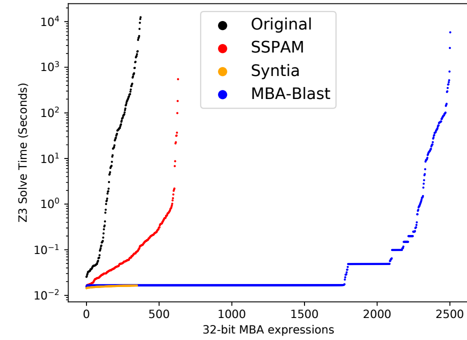
Figure 9 shows the procedure of using MBA-Blast to simplify the MBA sample in Figure 1.



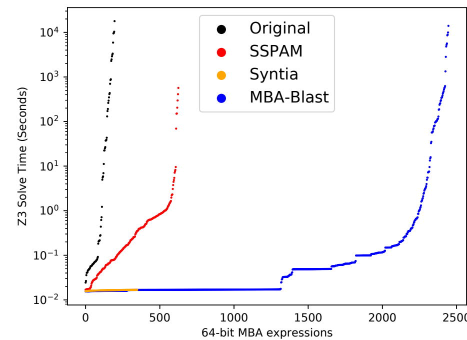
(a) 8-bit result.



(b) 16-bit result.



(c) 32-bit result.



(d) 64-bit result.

Figure 8: Compare Z3 solving time when handling all different data length in Dataset 2.

https://t.me/learningnets

$$
\begin{aligned}
&4*(\neg x \wedge y)-(x \oplus y)-(x \vee y)+4*\neg(x \vee y)-\neg(x \oplus y)-\neg y-(x \vee \neg y)+1+6*x+5*\neg z+(\neg(x \oplus z))-(x \vee z)-2*\neg x \\
&\qquad -4*(\neg(x \vee z))-4*(x \wedge \neg z)+3*(\neg(x \vee \neg z)) \\
&=4*(y-(x \wedge y))-(x+y-2*(x \wedge y))-(x+y-(x \wedge y))+4*(-x-y+(x \wedge y)-1)-(-x-y+2*(x \wedge y)-1)-(-y-1) \\
&\quad -(-y+(x \wedge y)-1)+1+6*x+5*(-z-1)+(-x-z+2*(x \wedge z)-1)-1*(x+z-(x \wedge z))-2*(-x-1) \\
&\quad -4*(-x-z+(x \wedge z)-1)+3*(z-(x \wedge z))-4*(x-(x \wedge z)) \\
&=4*y-4*(x \wedge y)-x-y+2*(x \wedge y)-x-y+(x \wedge y)-4*x-4*y+4*(x \wedge y)-4+x+y-2*(x \wedge y)+1+y+1+y- \\
&\quad (x \wedge y)+1+1+6*x-5*z-5-x-z+2*(x \wedge z)-1-x-z+(x \wedge z)+2*x+2+4*x+4*z-4*(x \wedge z)+4+3*z \\
&\quad -3*(x \wedge z)-4*x+4*(x \wedge z) \\
&=x+y
\end{aligned}
$$

Figure 9: MBA-Blast simplification procedure of the example in Figure 1.

https://t.me/learningnets