Boosting SMT Solver Performance on Mixed-Bitwise-Arithmetic Expressions

Dongpeng Xu
University of New Hampshire, USA
dongpeng.xu@unh.edu

Binbin Liu*
University of Science and Technology of China, China
University of New Hampshire, USA
binbin.liu@unh.edu

Weijie Feng
University of Science and Technology of China, China
fengwj@mail.ustc.edu.cn

Jiang Ming
University of Texas at Arlington, USA
jiang.ming@uta.edu

Qilong Zheng
Jing Li
University of Science and Technology of China, China
{qlzheng,lj}@ustc.edu

Qiaoyan Yu
University of New Hampshire, USA
qiaoyan.yu@unh.edu

Abstract

Satisfiability Modulo Theories (SMT) solvers have been widely applied in automated software analysis to reason about the queries that encode the essence of program semantics, relieving the heavy burden of manual analysis. Many SMT solving techniques rely on solving Boolean satisfiability problem (SAT), which is an NP-complete problem, so they use heuristic search strategies to seek possible solutions, especially when no known theorem can efficiently reduce the problem. An emerging challenge, named Mixed-Bitwise-Arithmetic (MBA) obfuscation, impedes SMT solving by constructing identity equations with both bitwise operations (and, or, negate) and arithmetic computation (add, minus, multiply). Common math theorems for bitwise or arithmetic computation are inapplicable to simplifying MBA equations, leading to performance bottlenecks in SMT solving.

In this paper, we first scrutinize solvers' performance on solving different categories of MBA expressions: linear, polynomial, and non-polynomial. We observe that solvers can handle simple linear MBA expressions, but facing a severe performance slowdown when solving complex linear and non-linear MBA expressions. The root cause is that complex MBA expressions break the reduction laws for pure arithmetic or bitwise computation. To boost solvers' performance, we propose a semantic-preserving transformation to reduce the mixing degree of bitwise and arithmetic operations. We first calculate a signature vector based on the truth table extracted from an MBA expression, which captures the complete MBA semantics. Next, we generate a simpler MBA expression from the signature vector. Our large-scale evaluation on 3000 complex MBA equations shows that our technique significantly boost modern SMT solvers' performance on solving MBA formulas.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: Mixed Boolean Arithmetic, SMT Solvers, Simplification

ACM Reference Format:
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 (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454068

1 Introduction

Satisfiability modulo theories (SMT) solvers have been widely adopted in various software engineering areas, such as software analysis [43], verification [11], symbolic/concolic execution [10, 25, 27, 38], and test-case generation [8, 22, 23, 41]. In general, these techniques abstract software behavior as certain forms of math formulas and then use an SMT solver to decide their satisfiability. How efficiently the SMT solver solves these formulas directly impacts the effectiveness of those methods.

As the basics of SMT solving, Boolean satisfiability problem (SAT) is the first proved NP-Complete problem [15], so a practical SMT solver implementation inevitably relies on

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

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454068

```
1 #!/usr/bin/python3
2 from z3 import *
3
4 x = BitVec('x', 64)
5 y = BitVec('y', 64)
6 solve(x*y != (x&~y)*(~x&y) + (x&y)*(x|y))
```

Figure 1. An MBA example impedes SMT solving. Z3 solver cannot return a result in 1 hour on a machine with quad-core 3.6GHz CPU and 64 GB RAM.

heuristic-based search. However, these heuristic strategies tend to be highly customized for groups of known problems (e.g., SMT-LIB benchmarks [3]), easily leading to poor performance on new problems not anticipated by solver developers [16]. This is a rising challenge because SMT solvers are applied to diverse new research areas in programming language, software engineering, and security.

One recent challenge comes from a newly proposed technique called Mixed Bitwise and Arithmetic (MBA) transformation [2]. It transforms a simple expression to a complex structure with mixed bitwise and arithmetic computation, still equivalent to the original expression. This technique has been adopted by multiple software protection products to obfuscate data flow relationship [13, 24, 28, 33, 39]. Recent study [17] has demonstrated that the Z3 solver [34] and mathematical software [31, 40, 45] fail to simplify complex MBA expressions. Please note that MBA expressions do not contain those well-known challenges for SMT solvers, such as floating-point computation [19, 29], high-order logic [32], string solving [21, 46], or crypto hash functions [42]. In contrast, MBA expressions only contain basic bitwise and arithmetic operations such as $\wedge, \vee, \neg, +, -, \times$. They are challenging to simplify because existing math reduction rules only work either on pure bitwise expressions (e.g., via normalization and constraint solving) or on pure arithmetic expressions (e.g., via arithmetic reduction). The mixed usage of bitwise and arithmetic computation invalidates the reduction heuristics in solvers, so the solving procedure resorts to expensive brute-force search heuristics, incurring a high performance penalty.

Figure 1 presents a hard-to-solve MBA equation written in Z3's Python interface. Lines 4 and 5 declare x and y as two 64-bit variables. Line 6 verifies an MBA identity equation. On our testing machine with quald-core 3.6GHz CPU and 64 GB RAM, Z3 cannot return a result in one hour.

In this paper, we first present a comprehensive study to evaluate the performance of state-of-the-art SMT solvers when solving MBA expressions. Our test cases include MBA expressions collected and synthesized from existing documents with diverse complexity categories: linear, polynomial, non-polynomial. These MBA formulas are fed to SMT solvers, and the solving time is measured. The testing result reveals that SMT solvers can solve simple linear MBA but suffers from severe performance overhead when solving complex linear, polynomial, and non-polynomial MBA. Enlightened by this finding, we design a novel, semantic-preserving transformation to translate complex MBA expressions to simple, easy-to-solve forms. The key idea is to translate complex MBA computation to a normalized MBA set, which has much less mixing bitwise and arithmetic operations. More specifically, we first calculate a *signature vector* to capture the essential semantics of an MBA expression. Then our method generates simple, normalized MBA expressions from the signature vector. Because the mixing degree of bitwise and arithmetic calculation is reduced in the simplified form, the math reduction laws inside SMT solvers are more applicable and hence solvers' performance is boosted.

Our study involves modern SMT solvers including Z3 [34], STP [20], and Boolector [9] on a large-scale corpus containing 3000 MBA expressions. The study result demonstrates that these solvers encounter performance problems on solving MBA expressions. They can only solve up to 15.6% of MBA expressions. We implement our MBA simplification method as a prototype named *MBA-Solver*. With the help of MBA-Solver, solvers achieve a significant performance improvement, solving 96.5% of MBA expressions.

Contributions. In a nutshell, our paper makes the following three key contributions.

- We investigate and reveal the performance problem when applying modern SMT solvers to MBA expressions. Our in-depth study shows that SMT solvers can solve simple linear MBA, but they are seriously impeded when solving more complex cases, such as polynomial and non-polynomial MBA.
- We develop a new technique to boost solvers' performance on solving MBA expressions. Our method introduce a semantic-preserving transformation to reduce complex MBA to a simple, easy-to-solve form. The key idea is to reduce the mixing degree of bitwise and arithmetic operations so that arithmetic reduction rules become effective again.
- We implement our method as a prototype called MBA-Solver and evaluate it with the state-of-the-art solvers. The result shows MBA-Solver can effectively reduce MBA expressions to easy-to-solve forms, increasing the percentage of solved MBA equations from 15.6% to 96.5%. Our implementation and datasets are publicly available in the artifact of this paper.

2 Background

To better explain our work, we first introduce the basics of MBA identity equations. Then we discuss the existing efforts on solving MBA, pointing out the limitations which also set up the research tasks in this paper.

2.1 Mixed Bitwise and Arithmetic

Generally speaking, an *MBA expression* mixes the usage of bitwise operations and integer arithmetic computation. Here is an example. Note that the bitwise expressions like $x \wedge y$ and $x \oplus y$ are used arithmetically as an addend or multiplicand.

$$x + 2y + x \wedge y - 3(x \oplus y) + 4 \tag{1}$$

MBA expressions can be used to construct an *MBA identity equation*, which means, for any input, the MBA expression on the left and right side of the equation are always equivalent. The example shown in Figure 1 in the introduction section is actually an MBA identity equation, for all x, y as integers. Note that the arithmetic operations in MBA are restricted on integers, because they can be encoded as bitvectors for bitwise operations.

Historically, MBA identity equations are known as programming tricks for solving specific problems or optimizing performance. For example, HAKMEM Memo [4] and Hacker's Delight [44] collect plenty of these equations. We show two examples as follows.

$$x \vee y = x \wedge \neg y + y \tag{2}$$

$$x \oplus y = x \vee y - x \wedge y \tag{3}$$

For all integers, these two MBA equations always hold. Equation (2) shows how to use $\wedge, \neg, +$ to calculate $x \vee y$. Similarly, Equation (3) calculates $x \oplus y$ using $\vee, -, \wedge$. Combine them together, we can calculate $x \oplus y$ using $+, -, \wedge, \neg$. Therefore, these MBA identity equation can be used for implementing new instructions from a basic instruction set, e.g., on a RISC instruction architecture.

MBA identity equations essentially expose the equivalence relationship between two MBA expressions. Inspired by this characteristic, Zhou et al. [47, 48] extend the existing MBA equations and generalize an abstract model named "Boolean-arithmetic algebras". In this model, the concept of MBA is specified as linear MBA and polynomial MBA, whose definitions are shown as follows. Their relation is shown in Figure 2.

Definition 1. *A linear MBA expression is:*

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

a_i is a coefficient (integer constant).
e_i is a bitwise expression of variables $x_1, x_2, \ldots, x_t$.
The bitwise operators include inclusive-or $\vee$, and $\wedge$, exclusive-or $\oplus$, and not $\neg$.
$a_i e_i$ is called a term.

Expression (1) is a linear MBA expression. It contains five terms: $x, 2y, x \wedge y, -3(x \oplus y), 4$. If the bitwise expression e_i is tautology `True`, then the term becomes a constant term only including the coefficient like 4.

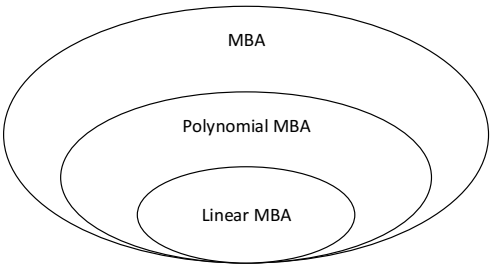

Figure 2. The relation between all MBA expressions, polynomial MBA, and linear MBA.

Definition 2. *A polynomial MBA expression is:*

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

a_i is a coefficient (integer constant).
$e_{i,j}$ are bitwise expressions of variables $x_1, \ldots, x_t$.
$a_i \big(\prod_{j \in J} e_{i,j}(x_1, \ldots, x_t) \big)$ is called a term.

Polynomial MBA extends linear MBA by allowing the multiplication of bitwise expressions in each term. In the following example, each term is the product of a coefficient and one or more bitwise expressions.

$$xy + 2(x \wedge y) + 3(x \wedge \neg y)(x \vee y) - 5 \tag{4}$$

With the formalized MBA definition, previous work [48] proposes a method to construct linear MBA identity equations. First, it enumerates all possible values in the truth table of every bitwise expression. Then it treats the truth table as the matrix of a linear equation system and tries to find a solution. Lastly, the solution is applied as the coefficients in MBA. We present a concrete example as follows to explain these steps in details.

Example 1. *We will build an MBA identity equation from the following truth table:*

$$M = \begin{array}{c} \begin{array}{ccccc} x & y & x \oplus y & x \vee \neg y & -1 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

The column with all "1" is encoded as -1 to guarantee it also works on 2's complement integers. Using the truth table as a matrix M, we can calculate an MBA identity equation that always equals to zero as follows, where $C_1, C_2, \ldots, C_5$ are coefficients.

$$C_1 x + C_2 y + C_3(x \oplus y) + C_4(x \vee \neg y) + C_5(-1) = 0 \tag{5}$$

Solving the following linear equation system:

$$M\vec{C} = 0, \vec{C} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{pmatrix}$$

produces the following solution.

$$\begin{cases} C_4 + C_5 = 0 \\ C_2 + C_3 + C_5 = 0 \\ C_1 + C_3 + C_4 + C_5 = 0 \\ C_1 + C_2 + C_4 + C_5 = 0 \end{cases} \Longrightarrow \begin{cases} C_1 = 1 \\ C_2 = -1 \\ C_3 = -1 \\ C_4 = -2 \\ C_5 = 2 \end{cases}$$

Substituting the solution to equation (5) produces an MBA identity equation.

$$x - y - (x \oplus y) - 2 * (x \vee \neg y) + 2 * (-1) = 0$$

So,

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

Note that MBA identity equation holds on n-bit 2's complement integers, which forming an integer modular ring $Z/(2^n)$. When the arithmetic calculation overflows, it starts from the minimum value. This is why the all "1" column is encoded as "-1". All bits being "1" encodes the boundary value in the integer modular ring. In MBA calculation, every bit is computed separately as the coefficient of 2^n. Because the identity equation holds on every bit, it also holds on the n-bit vector. The formal math proof can be found in the MBA algebra work [48].

This method is a generic approach for generating non-trivial, complex expressions involving bitwise and integer operations. It lays the theoretical foundation of building MBA identity equations.

2.2 MBA Application

MBA identity equations expose the equivalence between two expressions, thus one direct application is software obfuscation, i.e., transforming a simple program into a complex form. For example, the MBA equation we built in Example 1 can be used for obfuscating $x - y$. Hacker's delight [44] collects numerous MBA equations. Eyrolles [17] and Banescu [1] implement and extend the above method to produce a collection of MBA identity equations. Here are some examples where the right part of each equation can be used as an obfuscated form of integer addition $x + y$.

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

Due to the solid theoretic foundation and implementation simplicity, MBA has been applied to many practical obfuscation tools. For example, Quarkslab [39], Cloakware [28], and Irdeto [24] include MBA obfuscation in their commercial products. Tigress [12], an academic C source code diversifier/obfuscator, encodes integer variables and expressions into complex MBA forms [13, 14]. Mougey and Gabriel [33] present a real-world MBA example found in an obfuscated Digital Rights Management (DRM) software system. Blazy and Hutin [6] integrate formally verified MBA obfuscation rules into the generated binaries by the CompCert C compiler [26]. Recently, Xmark adopted MBA obfuscation to conceal the static signatures of software watermarking [30]. ERCIM News also reported in 2016 that MBA obfuscation has been detected in malware compilation chains [5].

Some preliminary investigations have shown that MBA obfuscated expressions are a big hurdle for people trying to understand a program. Eyrolles [17] applies multiple simplification methods (e.g., mathematical reduction, compiler optimization, and solver simplification) to complex MBA expressions, but cannot effectively produce simplified results. Bardin et al. present a novel technique in IEEE S&P'17 to assist obfuscated binary analysis, called *backward-bounded dynamic symbolic execution* [2]. However, the authors admitted that MBA obfuscation introduces hard-to-solve predicates and is a major obstacle to their approach [36, 37]. In our experiment, the state-of-the-art SMT solver Z3 [34] fails to return any result in one hour, when solving the MBA identity equation presented in Figure 1. These preliminary results inspire us to conduct an in-depth study to investigate SMT solvers' performance on MBA equations.

3 Study of Solvers' Performance on Solving MBA Equations

This section reports our comprehensive study of solvers' performance on solving MBA equations. Our goal is to fully expose the capability of SMT solvers on solving diverse MBA equations. More specifically, we seek for answers to the following research questions (RQs).

1. **RQ1:** How much time do SMT solvers spend on solving a MBA equation?
2. **RQ2:** For different categories of MBA equations, how different is the solving time?
3. **RQ3:** What are the key factors that affect the solving time?

The best way to answer these questions is to run state-of-the-art SMT solvers on a large scale of diverse MBA equations and observe their solving performance. The rest of this section describes our experiment and result in details.

3.1 Experiment Setup

MBA Equation Corpus. We collect a large corpus including all MBA equations from existing books, research papers,

Table 1. The complexity distribution of the MBA corpus. Poly MBA is short for non-linear polynomial MBA.

Metrics	Linear MBA			Poly MBA			Non-poly MBA		
	Min	Max	Average	Min	Max	Average	Min	Max	Average
Num of Variables	1	4	2.5	1	4	2.4	1	4	2.9
MBA Alternation	0	14	9.1	4	15	9.1	0	44	17.2
MBA Length	4	362	116.5	25	307	88.0	6	478	161.6
Number of Terms	2	14	9.8	2	13	7.4	2	49	17.1
Coefficients	1	35	7.2	1	294	16.0	1	293	22.1

technical report, and other documents. We collect 3, 000 MBA from the following sources:

1. Syntia [7]. It adopts program synthesis for synthesizing obfuscated code's semantics. The authors use MBA samples to evaluate their methods. These MBA samples are public available on GitHub.
2. Eyrolles's PhD thesis [17]. Eyrolles's work is one of the earliest research works on MBA. Her thesis includes many MBA samples and also extends the original MBA generation method.
3. Tigress [12]. It is an automated software obfuscation tool which translates normal arithmetic expressions into obfuscated MBA expressions.
4. MBA obfuscation papers [47, 48]. These papers include numerous MBA samples to illustrate the effectiveness of MBA obfuscation.
5. Hacker's Delight [44] and HAKMEM memo [4]. These documents include MBA samples for software optimization.

MBA Complexity Metrics. To anatomize how MBA complexity affect the solving performance, we measure MBA complexity in terms of the following metrics. Table 1 shows the distribution of these metrics.

1. **MBA Type.** The MBA expression is linear, poly, or non-polynomial. To avoid ambiguity, poly MBA in the rest sections of this paper refers to "non-linear" polynomial MBA.
2. **Number of Variables.** How many variables are involved in the MBA expression.
3. **MBA Alternation.** The number of operations that connect arithmetic and bitwise operations. For example, in $(x \wedge y) + 2z$, the + represents an MBA alternation operation, because its left operand is a bit-vector generated by $(x \wedge y)$, and its right operand is an integer arithmetic $2z$.
4. **MBA Length.** Considering an MBA expression as a character string, the string length is measured as the MBA length.
5. **Number of Terms.** How many terms are included in an MBA expression.
6. **Coefficient.** How large the coefficients are in every terms.

SMT Solvers. We test three popular SMT solvers: Z3 [34], Boolector [9], and STP [20]. Z3 and STP are broadly used in software analysis and verification. Boolector is the winner in SMT-COMP [35]. All these solvers support Python interface, which enables us to conveniently generate MBA expressions in normal Python syntax. In our experiment, the solving timeout threshold is set as 1 hour.

Machine Configuration. Our experiments are performed on a server with the following configuration.

- CPU: Intel Xeon W-2123 4-Core 3.60GHz
- Memory: 64GB 2666MHz DDR4 RAM
- Hard Drive: 2.5TB SSD
- Operating system: Ubuntu 18.04

3.2 Experiment Result

Overall, the testing result shows that all three solvers can only solve a small portion (less than 16%) of the MBA equations in the corpus. Z3 and STP solved less than 3% of the MBA, and Boolector performs better, solving 15.6% of the MBA. On average, Z3 and STP spend more than 300 seconds solving each test case. Boolector is faster, spending 183 seconds for each case on average. Figure 4 presents the solving time distribution of the three solvers. It shows that the solving time drastically grows when handling different MBA equations. All of three solvers do not return a result for majority MBA equations within the time threshold (1 hour). This result demonstrates that SMT solvers indeed suffer from performance bottleneck on solving MBA expressions.

We further compare the solving time of different categories of MBA equations and the result is shown in Table 2. The interesting finding is that, while SMT solvers can handle simple linear MBA equations, they suffer from heavy performance problems on non-linear MBA (poly MBA and non-poly MBA). As shown in Table 2, Boolector can solve 467 linear MBA equations in an average time of 184 seconds for each one. For non-linear MBA equations, only 29 can be solved.

To more precisely understand which complexity factors affect the solving performance, we carefully analyze and compare the solving time with different metrics. The analysis result is shown in Figure 3. We observe that MBA alternation is the dominant factor influencing solvers' performance. In

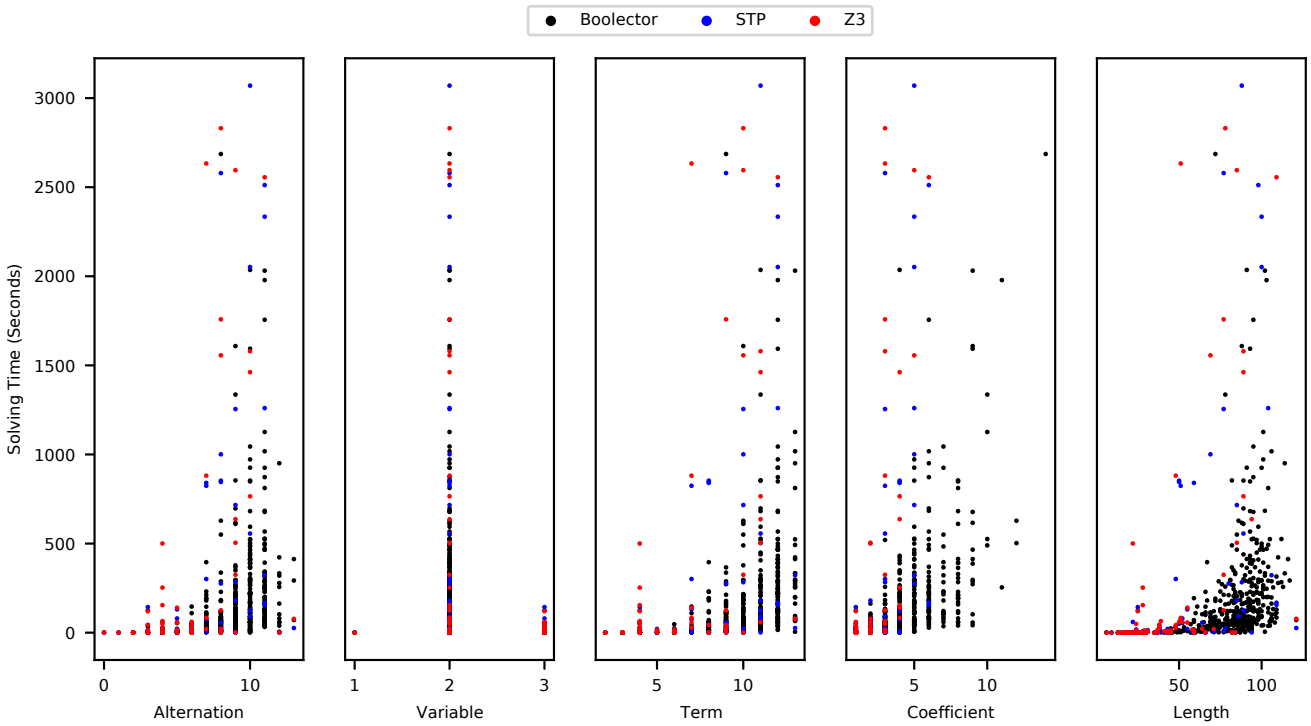

Figure 3. The relation between various metrics and solver's performance.

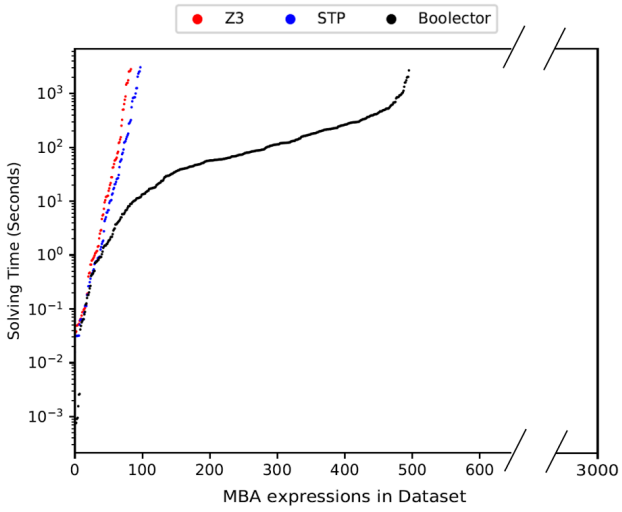

Figure 4. Solver's performance on our MBA testing dataset.

Figure 3, solving time drastically increase when the MBA alternation number grows. This observation demonstrates that the solving difficulty comes from the heterogeneous mixture of bitwise and arithmetic operations in MBA equations.

3.3 Conclusion

This MBA solving study helps us gain a clear picture of modern SMT solvers' performance on solving MBA identity equations. We summarize the most important findings as follows.

1. Solvers can solve simple linear MBA equations.
2. The solving difficulty resides in complex linear MBA, poly MBA, and non-poly MBA.
3. MBA alternation is the critical factor impairing SMT solving.

These interesting findings enlighten us to search for a semantic preserving transformation that reduces the MBA alternation in MBA expressions.

4 MBA Simplification

Enlightened by the study result, we develop a new MBA simplification method. The key insight is to design a *semantic preserving transformation to reduce MBA alternation*, because our study has shown that: 1) solvers can process simple MBA equations; 2) MBA alternation is the dominant factor for solving time. If we translate a complex MBA expression into a equivalent form with less MBA alternation, SMT solvers will have a higher chance to solve it.

Following this idea, we inspect the MBA design as shown in Section 2.1 and find a semantic-equivalent transformation to reduce MBA alternation. First, we extract a *signature vector*, which uniquely captures the mathematical semantics of an MBA expression. Our analysis proves that any MBA

Table 2. The experiment result of each SMT solver solving different categories of MBA equations. N is the number of solved MBA equations. $[T_{\min}, T_{\max}]$ presents the range from minimum to maximum solving time. T_{Avg} is the average solving time. Solving time is measured by seconds.

MBA Type	Z3			STP			Boolector		
	N	$[T_{\min}, T_{\max}]$	T_{Avg}	N	$[T_{\min}, T_{\max}]$	T_{Avg}	N	$[T_{\min}, T_{\max}]$	T_{Avg}
Linear MBA	55	[0.0, 2831.1]	394.7	69	[0.0, 3070.2]	336.6	467	[0.0, 2686.4]	184.0
Poly MBA	1	[0.1, 0.1]	0.1	1	[0.1, 0.1]	0.1	1	[0.1, 0.1]	0.1
Non-poly MBA	28	[0.1, 120.4]	16.1	28	[0.1, 143.3]	17.2	28	[0.0, 124.1]	15.0
Total Solved Number	84 (2.8%)			98(3.3%)			496 (16.5%)		

expressions with the same signature vector are always equivalent. The next step is to reduce the MBA alternation. We achieve this by transforming MBA expressions to a set of normalized expressions, whose MBA alternation is much lower than the original expressions.

Different from previous black-box methods such as pattern matching, fuzzing, and machine learning, our method demystifies MBA's mathematical mechanism, so it's directly built on the solid theoretical foundation. The simplification procedure is pure semantic preserving transformation, so it does not introduce false positives or false negatives. The rest of this section provide a detailed description of our method.

4.1 MBA Signature Vector

As the basics, we first introduce a concept called *signature vector*, which can represent the semantics of a group of equivalent linear MBA expressions. This concept derives from the MBA design in § 2.1. The principle is to encode all MBA calculation into an integer vector, so that two linear MBA expressions are equivalent, if and only if their signature vectors are equivalent. Therefore, for every linear MBA expression, we may calculate its signature vector and use it for deriving more equivalent MBA expressions.

Precisely, we define the signature vector as follows. Example 2 shows the procedure of calculating the signature vector for $2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$.

Definition 3. *The signature vector $\vec{s}$ of an linear MBA expression is the product of the MBA truth table matrix M and the coefficients vector v.*

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

Example 2. *For linear MBA expression $E = 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$,*

$$M = \begin{array}{c} \begin{matrix} x \vee y & \neg x \wedge y & x \wedge \neg y \end{matrix} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \end{array}, \vec{v} = \begin{pmatrix} 2 \\ -1 \\ -1 \end{pmatrix}$$

$$\vec{s} = M\vec{v} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix}$$

Based on this definition, for any two linear MBA expressions, if their signature vectors are equivalent, the product of their truth table and coefficients are equivalent. Because their truth tables enumerate all possible cases, it further implies that, for any input to the two MBA expressions, the calculation result is always the same. Theorem 1 presents an accurate proof showing that two signature vectors are equivalent, if and only if the two MBA expressions are equivalent. Hence signature vectors capture the essential semantics of an linear MBA expression. More importantly, it paves the way to simplifying more complex MBA expressions.

Theorem 1. *Given two linear MBA expressions E_1 and E_2,*

$$E_1 = E_2, \textit{ if and only if the signature vectors } \vec{s}_1 = \vec{s}_2$$

Proof. First prove sufficiency. If $\vec{s}_1 = \vec{s}_2$,

$$\because \vec{s}_1 = M_1\vec{v}_1,\ \vec{s}_2 = M_2\vec{v}_2$$

M_1 and M_2 are the truth table matrices. $\vec{v}_1$ and $\vec{v}_2$ are coefficient vectors.

$$\therefore M_1\vec{v}_1 = M_2\vec{v}_2$$

$$M_1\vec{v}_1 - M_2\vec{v}_2 = 0$$

$$[M_1\ M_2]\begin{pmatrix} \vec{v}_1 \\ -\vec{v}_2 \end{pmatrix} = 0$$

$$E_1 - E_2 = 0$$

$$E_1 = E_2$$

The necessity is proved by reversing the steps for sufficiency. □

4.2 Generating MBA from Signature Vectors

The benefit of calculating signature vectors is they are handy for generating equivalent MBA expressions. By linear algebra knowledge, we know that any vector can be represented by a linear combination of a set of base vectors. This math attribute is also applicable to signature vectors. As shown

Table 3. The truth table of bitwise expressions corresponding to the base vectors.

x	y	$\neg x \wedge \neg y$	$\neg x \wedge y$	$x \wedge \neg y$	$x \wedge y$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

below, the signature vector $\vec{s} = \begin{pmatrix}0\\1\\1\\2\end{pmatrix}$ can be split to the sum of four base vectors.

$$\begin{pmatrix}0\\1\\1\\2\end{pmatrix} = 0 * \begin{pmatrix}1\\0\\0\\0\end{pmatrix} + 1 * \begin{pmatrix}0\\1\\0\\0\end{pmatrix} + 1 * \begin{pmatrix}0\\0\\1\\0\end{pmatrix} + 2 * \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

It indicates that, if we can find those bitwise expressions corresponding with the base vectors, then we will build a new, equivalent MBA expression from the linear combination! All we need is to treat the base vectors as truth values in a truth table and construct the bitwise expressions. Table 3 shows such expressions and the corresponding truth table. Substituting the base vectors with the bitwise expressions, we get a new MBA expression E':

$$E' = (\neg x \wedge y) + (x \wedge \neg y) + 2 * (x \wedge y)$$

Because E' and E in Example 2 have the same signature vector: $\begin{pmatrix}0\\1\\1\\2\end{pmatrix}$. As proven by Theorem 1, the two MBA expressions are equivalent.

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

4.3 Reducing MBA Alternation

The signature vector and base vectors provide a semantic-preserving way to represent any linear MBA expressions by a linear combination of base vectors. We further notice that this feature can be used for reducing the MBA alternation in an MBA expression, if an appropriate set of base vectors are selected.

The idea is to minimize the number of bitwise expressions in the base vectors. Because the result is a linear combination of base vectors, the operations connecting those base vectors are arithmetic operations, i.e., $\times, +, -$. If the expression corresponding with the base vector is only one integer variable or constant, then there will be no MBA alternation around it. Therefore, our goal is to find a set of base vectors that has the *minimum* number of bitwise expressions.

Table 4. Normalized base vectors to reduce MBA alternation.

x	y	$x \wedge y$	-1
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	1

Table 4 shows such a set of base vectors for two-variable linear MBA. The corresponding base expressions are x, y, $x \wedge y$, and -1, where only one is bitwise expression.

When using these base vectors to represent the signature vector of Example 2: $\begin{pmatrix}0\\1\\1\\2\end{pmatrix}$, we need to solve the following linear equation system.

$$\begin{pmatrix}0\\1\\1\\2\end{pmatrix} = C_1 * \begin{pmatrix}0\\0\\1\\1\end{pmatrix} + C_2 * \begin{pmatrix}0\\1\\0\\1\end{pmatrix} + C_3 * \begin{pmatrix}0\\1\\1\\1\end{pmatrix} + C_4 * \begin{pmatrix}1\\1\\1\\1\end{pmatrix}$$

The result is

$$\begin{cases} C_1 = 1 \\ C_2 = 1 \\ C_3 = 0 \\ C_4 = 0 \end{cases}$$

Therefore,

$$\begin{pmatrix}0\\1\\1\\2\end{pmatrix} = \begin{pmatrix}0\\0\\1\\1\end{pmatrix} + \begin{pmatrix}0\\1\\0\\1\end{pmatrix}$$

Substituting the base vectors with the expressions in Table 4 produces the following simplification result. Note that the simplification eliminates all bitwise expressions, producing a pure arithmetic expression $x + y$. It reduces the MBA alternation from 3 to 0. Because MBA alternation is the key factor impeding SMT solving, this simplification will improve solvers' performance of processing MBA expressions.

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

4.4 Simplifying Non-linear MBA

The method we introduced so far can translate a complex linear MBA to a simple one with low MBA alternation. As indicated in our MBA study, non-linear MBA (poly MBA and non-poly MBA) placed the major challenges for SMT solvers. Thus we further investigate how to use this method to simplify non-linear MBA expressions.

In principle, we still focuses on reducing the degree of MBA alternation, because it is the major barrier to SMT solving. More specifically, we first simplify the sub-expressions in non-linear MBA and then expose more simplification opportunities. We generates a pre-computed mapping table to enumerate the signature vectors and the corresponding normalized MBA expressions. Table 5 illustrates such an example table for two-variable MBA expressions. The first four rows are the base vectors. Other rows are derivative ones, which are generated by solving linear systems as in Section 4.3.

Using the mapping table, we can transform any bitwise sub-expression in a non-linear MBA, to a normalized and simple form. Note that the signature vector column enumerates all possible truth values involving two variable. Therefore, for any bitwise expressions in non-linear MBA, we first calculate its signature vector, look it up in the mapping table, and substitute it with the simplified MBA. The substitution result is an MBA expression including only one type of bitwise expression: $x \wedge y$, so the MBA alternation is largely reduced. Traditional math rules can be further applied to merge terms and produce a concise form.

The following procedure shows how to use Table 5 to simplify the poly MBA example in the Introduction section. First, we calculate the signature vector of $x \wedge \neg y$ and the result is $\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$. Looking it up in Table 5, the simplified MBA expression is $x - x \wedge y$. Similarly, $\neg x \wedge y$ is transformed to $y - x \wedge y$ and $x \vee y$ is transformed to $x + y - x \wedge y$. The substitution and simplification procedure is shown as follows.

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

4.5 Optimization

MBA-Solver also includes several optimization methods to further improve the simplification performance.

Look-up table. Caching the simplification transformation in a look-up table can improve MBA-Solver's performance, because it saves us from re-computing the signature vector for every MBA expression. Also, the same sub-expression may appear multiple times when simplify an MBA expression.

Table 5. A pre-computed simplification table for two-variable MBA expressions. The first four rows are the base vectors and the rest rows enumerate other derivative signature vectors and show the corresponding MBA expressions. Signature vectors are shown horizontally for better presentation.

Type	Signature Vector	MBA Expression
Base	(0, 0, 1, 1)	x
	(0, 1, 0, 1)	y
	(0, 0, 0, 1)	$x \wedge y$
	(1, 1, 1, 1)	-1
Derivative	(0, 0, 0, 0)	0
	(0, 0, 1, 0)	$x - (x \wedge y)$
	(0, 1, 0, 0)	$y - (x \wedge y)$
	(0, 1, 1, 0)	$x + y - 2 * (x \wedge y)$
	(0, 1, 1, 1)	$x + y - (x \wedge y)$
	(1, 0, 0, 0)	$-x - y + (x \wedge y) - 1$
	(1, 0, 0, 1)	$-x - y + 2 * (x \wedge y) - 1$
	(1, 0, 1, 0)	$-y - 1$
	(1, 0, 1, 1)	$-y + (x \wedge y) - 1$
	(1, 1, 0, 0)	$-x - 1$
	(1, 1, 0, 1)	$-x + (x \wedge y) - 1$
	(1, 1, 1, 0)	$-(x \wedge y) - 1$

Common Sub-expression. During the simplification procedure, common sub-expressions can be replaced by a intermediate variable to expose further simplification opportunities. One example is shown as follows. After the first round of expression substitution, $x - y$ is shown as the common sub-expression. By replacing $x - y$ with an temporary variable t, the MBA expression can be further simplified. At the last step, t is substituted back with $x - y$ to produce the final result.

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

This strategy is especially effective when handling non-poly MBA, because non-poly MBA expressions are usually generated by applying MBA transformation to different parts of an expression. The common sub-expression optimization actually reverses this procedure.

Final-step optimization. This optimization is performed at the last step of the simplification to further reduce MBA alternation. Our simplification result only include $x, y, x \wedge$

Algorithm 1 MBA-Solver Algorithm.

```
1:  Input: MBA expression E and look-up table T
2:  function MBASolver(E)
3:      if E ∈ linear MBA then
4:          E' ← ApplyTransTable(E, T)
5:          for e ∈ E' ∧ e ∉ T do
6:              s⃗ ← SignatureVector(e)
7:              e' ← GenerateMBA(s⃗)
8:              Update(E', e')
9:          end for
10:         ArithReduce(E')
11:         FinalOptimize(E')
12:         return E'
13:     else
14:         for e ∈ SubExpr(E) do
15:             e' ← MBASolver(e)
16:             E' ← ReplaceExpr(E, e, e')
17:             ArithReduce(E')
18:             CommonSubExprOpt(E')
19:             FinalOptimize(E')
20:             return E'
21:         end for
22:     end if
23: end function
```

$y, 1$, which may not be the optimal form for some expressions, especially for pure bitwise expressions. For example, $x + y - 2(x \wedge y)$ actually can be further simplified to $x \oplus y$ so that MBA alternation further decrease from 1 to 0.

To produce the optimal result, at the last step, MBA-Solver will calculate the signature vector of the MBA and then try to replace it with the one bitwise operation. Note that this optimization is only performed at the final step, because using it in the middle will introduce extra bitwise expressions and impede the simplification effectiveness.

4.6 MBA-Solver Algorithm

We integrate the simplification and optimization methods into Algorithm 1. The algorithm takes an MBA expression E and a look-up table T as the input and returns its simplified form E'. First, it check the MBA expression is a linear MBA or not. For linear MBA, the algorithm applies the rules in the look-up table T. For those expressions not in T, it computes the signature vector and generate the normalized MBA. Then, an arithmetic reduction and the final optimization are performed before returning the simplification result. For non-linear MBA, the algorithm is applied to each sub-expression and replace it with the simplified result. It also invokes the common sub-expression optimization for further simplification. At last, it performs the final-step optimization and returns the simplification result.

5 Implementation

We implemented the MBA simplification method as a program analysis prototype called MBA-Solver. It is designed as a preprocessing pass before the MBA equation is passed to an SMT solver. The strength of this design is that our simplification can serve as a separate preprocessing pass without changing SMT solvers. It takes a complex MBA as the input and outputs the simplified result , which can be further solved by any SMT solver. An overview of MBA-Solver's architecture is shown in Figure 5.

Inside MBA-Solver, the total simplification consists of four major components. First, a parser reads the MBA equation and translates it to Abstract Syntax Tree (AST). Second, a tree substitution module substitutes proper expressions with normalized MBA expressions. The substitution is either directly derived from the look-up table, or produced by calculating the signature vector and then solving a linear equation system. A math arithmetic simplification is further performed to reduce the complexity. Then the optimization module performs common sub-expression optimization and final optimization. The last step is to translate the result from AST to an MBA formula.

The whole prototype is written in around 1800 lines of Python code. We leverage the `SymPy` library for math reduction, and `NumPy` library for matrix-vector product and solving linear equation system. MBA-Solver uses the Python interface to communicate with SMT solvers. MBA-Solver also includes utilities for measuring the MBA metrics such as counting MBA alternation.

6 Evaluation

In this section, we conduct experiments to evaluate MBA-Solver's effectiveness on boosting SMT solvers' performance when solving MBA equations. We are particularly interested in answering the following research questions (RQs).

1. **RQ1:** Compare with solving the original MBA expressions, how much performance improvement can MBA-Solver bring to SMT solvers?
2. **RQ2:** How is MBA-Solver's simplification result comparing with its peer tools?
3. **RQ3:** How much overhead does MBA-Solver introduce?

To answer these questions, we first run MBA-Solver and the peer tools on the same dataset as the MBA solving study in § 3, and then pass the simplification result to the three SMT solvers. As the answer to RQ1, we compare the number of solved MBA equations and the solving time with the ones in § 3. To address RQ2, we compare MBA-Solver's simplification result and SMT solving time with the peer tools. For RQ3, we report MBA-Solver's performance data such as execution time and memory usage.

Peer Tools for Comparison. We compare MBA-Solver with two state-of-the-art peer tools for MBA simplification:

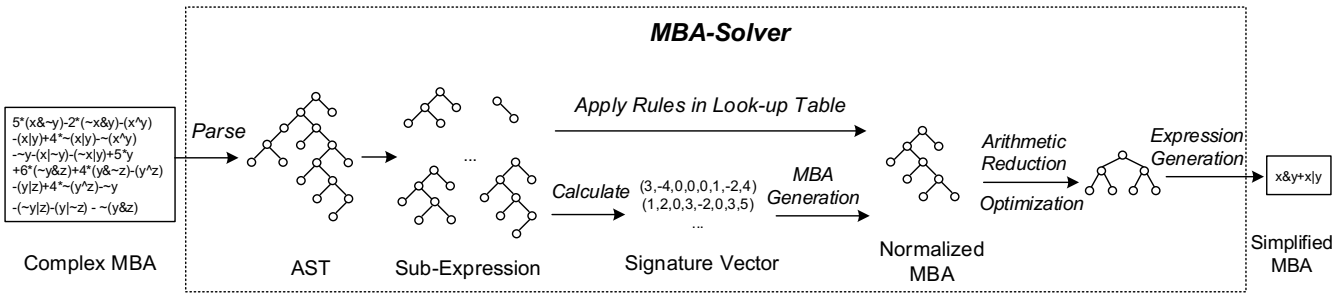

Figure 5. An overview of MBA-Solver's architecture.

SSPAM [18] and Syntia [7]. We download their latest open source version from GitHub. SSPAM (Symbolic Simplification with PAttern Matching) simplifies MBA expressions by pattern-matching. 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, including MBA expressions. It produces input-output pairs from instruction traces and then synthesizes a code snippet's semantic based on these input-output pairs. These two tools represent the state-of-the-art solutions from recent work on MBA simplification.

6.1 Boost SMT Solving MBA Equation

We apply MBA-Solver to the MBA equation corpus in § 3 and output the simplification result to the three SMT solvers. The evaluation result is presented in Table 6. Comparing it with Table 2, we observe a significant performance boost. After MBA-Solver's simplification, all three solvers successfully solve over 95% of MBA equations, while originally the solvers can only solve a small portion (Z3 2.8%, STP 3.3%, Boolector 16.5 %). An interesting finding is that Boolector's performance was notably better than the other two, but after MBA-Solver's simplification, the distinction becomes insignificant. It indicates that MBA-Solver simplification is a generic method, rather than over-fitting to a certain searching heuristic strategy or one specific solver.

Inspecting the detailed result, we find that all 1000 linear MBA and 1000 polynomial MBA equations are solved in a very short time (average solving time is 0.1 second). We also manually checked the simplification result and verify that the transformation in MBA-Solver effectively reduced linear and poly MBA to a normalized and concise form. For the majority of the non-poly MBA (894 out of 1000), MBA-Solver successfully reduce their complexity to a solver-friendly level (before MBA-Solver's simplification, only 28 of them can be solved).

We also manually check those non-poly MBA equations that cannot be solved. The main reason is that the non-poly MBA category contains numerous ad-hoc cases that escape from the normalization model in MBA-Solver. One such exception is when non-poly MBA includes sub-expression $\neg(x-1)$. MBA-Solver cannot reduce it because $x-1$ is already the normalized form. However, the correct simplification result is $\neg(x-1) = -x$. Even for this exception, MBA-Solver still successfully processed several cases by treating $x-1$ as a common sub-expression. Overall, non-poly MBA is not a well-defined model. Any MBA expression is considered as a non-poly MBA, if it does not satisfy Definition 2. At last, MBA-Solver demonstrates its capability by successfully simplifying 89.4% of them.

6.2 Comparison with Peer Tools

In this experiment, we run SSPAM and Syntia on the MBA equation corpus and send their output for SMT solving. The evaluation result is shown in Table 7 and Figure 6.

Correctness. As the peer tools, SSPAM relies on pattern matching and Syntia employs program synthesis. These techniques are not necessarily semantic-preserving transformation, which may produce incorrect simplification result. Therefore, we first investigate and compare the correctness. Note that all equations in the corpus are MBA identity equations, so the equivalence checking result from solvers must be "Yes". If a solving result is nonequivalent, then the simplification result must be wrong. Besides, the solver certainly may not be able to return the solving result due to MBA's complexity, so those cases are counted as timeout. The "Number of Correctness" column in Table 7 present the correctness result. SSPAM does not introduce wrong simplification result, because their pattern matching rules are semantic-equivalent. However, only 89 MBA equations are solvable after SSPAM's simplification. Solvers still experience timeout on the remaining 2911 MBA equations. The reason is due to the limited number of cases in SSPAM's pattern library.

Syntia simplification employs stochastic program synthesis, which approximates program semantics using Monte Carlo Tree Search (MCTS). Its effectiveness 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-Solver. In our experiment, Syntia can output a simplified expression for every MBA

Table 6. SMT solving time on MBA-Solver's simplification result. N is the number of solved MBA equations. $[T_{\min}, T_{\max}]$ presents the range from minimum to maximum solving time. T_{Avg} is the average solving time. All time is measured by seconds.

MBA Type	Z3			STP			Boolector		
	N	$[T_{\min}, T_{\max}]$	T_{Avg}	N	$[T_{\min}, T_{\max}]$	T_{Avg}	N	$[T_{\min}, T_{\max}]$	T_{Avg}
Linear MBA	1000	[0.01, 0.03]	0.02	1000	[0.01, 0.03]	0.02	1000	[0.01, 0.01]	0.01
Poly MBA	1000	[0.01, 0.04]	0.02	1000	[0.01, 0.04]	0.02	1000	[0.01, 0.02]	0.01
Non-poly MBA	894	[0.01, 53.9]	0.21	894	[0.01, 46.7]	0.20	894	[0.01, 43.9]	0.18
Total Solved Number	2894 (96.5%)			2894(96.5%)			2894 (96.5%)		

Table 7. Comparing MBA-Solver simplification result with the peer tools. In "Number of Correctness" column, "Y" means equivalent, "N" means not equivalent, and "O" means time out (solvers fails to return a result in one hour), and "Ratio" indicates the ratio of outputs passing equivalence checking. "Average MBA Alternation" and "Average Solving Time" only report the result of *correctly* simplified samples. "Before" and "After" means before and after the simplification. "Average Solving Time" reports the average time that a solver takes for solving one MBA sample.

Tools	Number of Correctness				Average MBA Alternation			Average Solving Time		
	Y	N	O	Ratio (%)	Before	After	A/B (%)	Z3	STP	Boolector
SSPAM	89	0	2911	3.0	4.8	4.3	89.6	242.35	231.9	156.6
Syntia	512	2488	0	17.1	3.3	0.4	12.1	0.03	0.03	0.02
MBA-Solver	2894	0	106	96.5	11.9	2.8	23.5	0.08	0.07	0.06

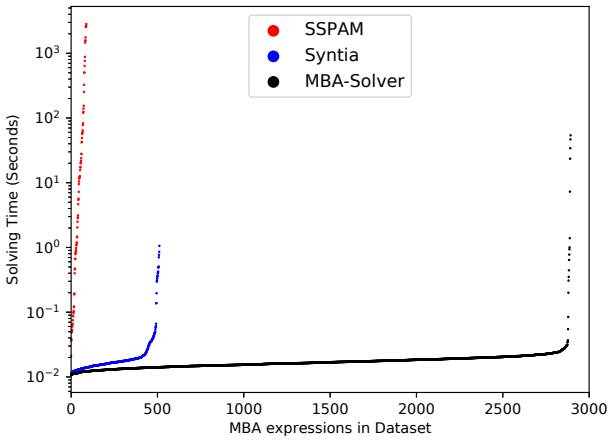

Figure 6. Z3 solving time with MBA-solver's simplification.

equation, but up to 82.9% of them are incorrect. For the rest of correct outputs, Syntia's effect rivals MBA-Solver, which is reflected by Figure 6.

In MBA-Solver's result, all 2894 simplified results are solved as equivalent, so it does not introduce any incorrect simplification result. This result demonstrates the semantic-preserving transformation in MBA-Solver.

Effectiveness. For all correct simplification results, the rest part of Table 7 compares their effectiveness. One interesting finding is that, SSPAM and Syntia can also handle simple MBA expressions well (average MBA alternation is around 3.0), but MBA-Solver can simplify much more complex MBA. MBA-Solver achieves the best simplification effectiveness. After the simplification, MBA alternation is only 23.5% of that before the simplification, which is the lowest among the peer tools. Moreover, the average solving time for MBA-Solver's simplification result is the fastest. It is because MBA-Solver significantly reduces MBA alternation, which is the key factor affecting the solving performance.

Table 8. MBA-Solver's performance on MBA expressions with different complexity.

Complexity of MBA Alternation	Time (Second)	Memory (MB)
10	0.05	0.2
20	0.68	1.5
30	0.79	3.6
40	0.93	6.7

6.3 Performance

This section reports MBA-Solver's performance data. Table 8 presents the time and memory cost when MBA-Solver process different complexity of MBA expressions measured by MBA alternation. MBA-Solver is very effective because it does not rely on any search or heuristic method. Our implementation adopts the Python `NumPy` library to efficiently perform matrix-vector product. Overall, MBA-Solver only introduces a negligible overhead.

7 Discussion

MBA-Solver has demonstrated the feasibility of simplifying MBA expressions. We also notice some potential enhancements that can be further investigated as future work.

Table 9. The truth table of another set of base vectors.

x	y	$x \vee y$	-1
0	0	0	1
0	1	1	1
1	0	1	1
1	1	1	1

Base vector selection. For the majority of the cases in our experience, using $(x, y, x \wedge y, -1)$ as the normalized base vector can effectively reduce MBA alternation and produce a concise result. However, some other cases can be simplified more effectively if using a different set of base vectors. Table 9 shows a set of base vectors different from the ones in Table 4.

In our observation, combing $x, y, -1$ with one bitwise expression usually produces a set of base vectors that reduce MBA alternation. Note that $x \wedge y$ is used in Table 4 and $x \vee y$ is used in Table 9. The optimal selection of that bitwise operation might depend on the specific input MBA expressions. A more in-depth analysis of the input MBA expression can be investigated to guide the selection of base vectors.

MBA mixing with other transformation. One possible threat to MBA-Solver is combining MBA transformation with other data encoding techniques to create complex expressions using bitwise and arithmetic operations. MBA-Solver is designed for solving MBA expressions, so it may correctly output some intermediate results for certain MBA sub-expressions, but cannot handle the remaining non-MBA part. It will be interesting to further investigate whether MBA-Solver can collaborate with other simplification techniques such as arithmetic reduction or machine learning to produce better results.

8 Conclusion

Mixed-Bitwise-Arithmetic (MBA) expression, which mixes both bitwise and arithmetic operations to generate an unintelligible expression, is an imminent threat to SMT solvers. The cost of applying MBA transformation is rather low, but the resulting expression becomes an insurmountable obstacle to both human analysts and SMT solvers. MBA obfuscation has been adopted by commercial software protection projects. The existing efforts to counter MBA obfuscation either work in an ad-hoc manner or suffer from unacceptable error rates. Our work is the first to study the SMT solving performance problem on MBA expressions and address this challenge by transfoming MBA expressions to easy-to-solve forms. We conduct a thorough study to reveal modern SMT solvers' performance on solving different types of MBA. We then investigate the underlying mechanism of MBA expression and represent the complete semantics of an MBA expression as a signature vector. Given a signature vector, we can calculate a linear combination of bitwise expressions to get a simplified version of the MBA expression. Our large-scale experiment with existing work demonstrates our prototype, MBA-Solver, is able to significantly boost SMT solvers' performance on solving MBA expressions. The impact of MBA-Solver can advance the application of SMT solving technique in automated software reverse engineering and malware analysis.

Acknowledgments

We would like to thank our shepherd Vijay Ganesh and the anonymous paper and artifact reviewers for their helpful feedback. We appreciate Rachael Little for proofreading the paper. This research was supported by NSF grant CNS-1948489. Jiang Ming was supported by NSF grant CNS-1850434.

References

[1] Sebastian Banescu and Alexander Pretschner. 2018. Chapter Five - A Tutorial on Software Obfuscation.

[2] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*. https://doi.org/10.1109/SP.2017.36

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard – Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*.

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

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

[6] Sandrine Blazy and Rémi Hutin. 2019. 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)*. https://doi.org/10.1145/3293880.3294103

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

[8] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. https://doi.org/10.1109/ICSE.2013.6606558

[9] Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. https://doi.org/10.1007/978-3-642-00768-2_16

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.

[11] Jianhui Chen and Fei He. 2018. Control Flow-Guided SMT Solving for Program Verification. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. https://doi.org/10.1145/3238147.3238218

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

Applications Conference (ACSAC'12). https://doi.org/10.1145/2420950.2420997

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

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

[15] Stephen A Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. 151–158. https://doi.org/10.1145/800157.805047

[16] Leonardo De Moura and Grant Olney Passmore. 2013. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics*.

[17] Ninon Eyrolles. 2017. *Obfuscation with Mixed Boolean-Arithmetic Expressions: Reconstruction, Analysis and Simplification Tools*. Ph.D. Dissertation. Université Paris-Saclay.

[18] Ninon Eyrolles, Louis Goubin, and Marion Videau. 2016. Defeating MBA-based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO'16)*. https://doi.org/10.1145/2995306.2995308

[19] Zhoulai Fu and Zhendong Su. 2016. XSat: A Fast Floating-Point Satisfiability Solver. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*. https://doi.org/10.1007/978-3-319-41540-$6_1$1

[20] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference in Computer Aided Verification (CAV'07)*. https://doi.org/10.1007/978-3-540-73368-$3_5$2

[21] Vijay Ganesh, Adam Kiezun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2011. HAMPI: A String Solver for Testing, Analysis and Vulnerability Detection. In *Proceedings of 23rd International Conference on Computer Aided Verification (CAV'11)*. https://doi.org/10.1007/978-3-642-22110-1_1

[22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation (PLDI'05)*. https://doi.org/10.1145/1065010.1065036

[23] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

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

[25] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15)*. https://doi.org/10.1145/2771783.2771806

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

[27] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving. In *Proceedings of 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*.

[28] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. 2008. A Compiler-based Infrastructure for Software-protection. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'08)*. https://doi.org/10.1145/1375696.1375702

[29] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. https://doi.org/10.1145/3338906.3338921

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

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

[32] Aleksandar Milicevic, Joseph P Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. https://doi.org/10.1007/s10703-016-0267-2

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

[34] Leonardo De Moura and Nikolaj Bjørner. 2008. 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)*. https://doi.org/10.1007/978-3-540-78800-$3_2$4

[35] Aina Niemetz, Mathias Preiner, and Armin Biere. 2016. *Boolector at the SMT Competition 2017*. Technical Report. Institute for Formal Models and Verification, Johannes Kepler University.

[36] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. 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)*. https://doi.org/10.1145/3359789.3359812

[37] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. 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)*. https://doi.org/10.1145/3371307.3371309

[38] Corina S Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic Execution with Mixed Concrete-Symbolic Solving. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)*. https://doi.org/10.1145/2001420.2001425

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

[40] sagemath. 2020. SageMath. http://www.sagemath.org/.

[41] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/1095430.1081750

[42] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[43] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. Search-Driven String Constraint Solving for Vulnerability Detection. In *Proceedings of 39th International Conference on Software Engineering (ICSE'17)*. https://doi.org/10.1109/ICSE.2017.26

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

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

[46] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-Based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. https://doi.org/10.1145/2491411.2491456

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

[48] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Proceedings of the 8th International Conference on Information Security Applications (WISA'07)*. https://doi.org/10.1007/978-3-540-77535-5_5