Simplification of General Mixed Boolean-Arithmetic Expressions: GAMBA [*]

Benjamin Reichenwallner & Peter Meerwald-Stadler
Denuvo GmbH
Salzburg, Austria

Abstract

Malware code often resorts to various self-protection techniques to complicate analysis. One such technique is applying Mixed-Boolean Arithmetic (MBA) expressions as a way to create opaque predicates and diversify and obfuscate the data flow.

In this work we aim to provide tools for the simplification of nonlinear MBA expressions in a very practical context to compete in the arms race between the generation of hard, diverse MBAs and their analysis. The proposed algorithm *GAMBA* employs algebraic rewriting at its core and extends *SiMBA* [19]. It achieves efficient deobfuscation of MBA expressions from the most widely tested public datasets and simplifies expressions to their ground truths in most cases, surpassing peer tools.

1 Introduction

Mixed Boolean-arithmetic (MBA) expressions, which have been introduced in the year 2006 by Zhou et al. [27], are a commonly used technique in code obfuscation. Their use in malware samples and various digital rights management (DRM) implementations is documented in the literature [16, 17, 25]; see [12] for a detailed analysis of MBA usage in malware. They are believed to be one of the strongest-known code obfuscation techniques [7]. In an effort to conceal secret information like data and algorithms, basic expressions like constants are transformed into mixed Boolean-arithmetic expressions that are semantically equivalent. This results in an artificial increase in code complexity to obfuscate the code and make it less comprehensible. It is typically assumed that the resultant complex expressions cannot be easily simplified back to their original form.

However, recent research [19] suggests that all linear MBAs can be solved in a straightforward way. This work extends this finding and contributes a practical algorithm for the simplification of nonlinear MBAs.

1.1 Prior work

Due to the fact that MBA expressions incorporate both logical and arithmetic operations that are not well compatible [6], they cannot be genuinely resolved using SAT solvers or mathematical tools that are intended to handle solely logical or arithmetic expressions, resp.

In recent years a significant number of tools specifically dedicated to their deobfuscation have been published [7]. They use various techniques such as pattern matching (e.g., SSPAM [6]), neural networks (e.g., NeuReduce [8]), bit-blasting (e.g., Arybo [10]), stochastic program synthesis (e.g., Stoke [22], Syntia [1] and Xyntia [15]), synthesis-based expression simplification (e.g., QSynth [4] and msynth [2]), as well as a family of algebraic methods (e.g. *MBA-Blast* [12], *MBA-Solver* [26], *MBA-Flatten* [13], *SiMBA* [19]).

[*] Accepted for presentation at the 2nd Workshop on Robust Malware Analysis (WoRMA'23), co-located with the 8th IEEE European Symposium on Security and Privacy, Delft, The Netherlands, July 3 - 7, 2023.

The first step in code analysis often is symbolic execution: locating and extracting code from a malware sample and turning it into an abstract syntax tree (AST) [1,4,24]. In this work, we are not concerned with the many complications involved in lifting binary code, identifying an expression, etc. due to anti-debug/-trace techniques and program analysis limitations. We assume to have a pair of expressions $(e, e^\star)$, where e is the complicated MBA expression and $e^\star$ the corresponding simple, semantically equivalent ground truth. The aim is to simplify e in order to facilitate program analysis – ideally back to $e^\star$. This setup is simpler compared to the scenario considered in program synthesis [1,2,4], where obfuscated code (including control-flow problems) is taken as an input.

Zhou et al. [27] were also the first to describe how to generate random **linear** MBAs which are equivalent to a – usually simple – target expression $e^\star$ via a solution of a randomly generated equation system based on the finding that a bitwise expression is fully determined by its values in the set $B = \{0, 1\}$ of zeros and ones. Another construction is an iterative application of rewriting rules from a given codebook, mapping simple expressions to more complex ones, see, e.g., [23]. To make MBAs more resistant to deobfuscation, additional encoding (e.g., using *Tigress* [3]), invertible functions or point functions [23,27] can be applied. These methods may create significant challenges for many simplification tools due to the potential introduction of large constants. Both generation approaches can in general also be used for generating **nonlinear** MBAs. Obviously, a codebook may also be used for the simplification of MBAs. While MBA expressions may not be immediately present in the codebook, an SMT solver can be utilized to check for equivalency against the simpler MBAs listed.

In 2021, Liu et al. [12,26] pointed out that Zhou et al.'s approach can be inverted too, pathing the way for surprisingly simple algebraic simplification of linear MBAs. This finding, which is actually already stated in Zhou et al.'s paper [27], but remained unnoticed due to a mistake, is leveraged by the simplifiers *MBA-Blast* [12], *MBA-Solver* [26], *MBA-Flatten* [13] and *SiMBA* [19] operating on **linear** MBAs. These tools outperform other existing tools significantly for this class of MBAs when it comes to simplification success and runtime.

For the analysis of binary code, interactive disassemblers and decompilers are used. The *SiMBA* algorithm has been integrated in Hex-Ray's *gooMBA* plugin [9] for *IDA Pro*, to help turn complex MBA expressions given as IR code into simple terms. In a practical setting, fast, correct, easy-to-use and potent expression simplification is highly desirable. Early algorithms tend to be too slow (*Arbyo*, *SSPAM*, *Syntia*), some require additional input (e.g., *MBA-Solver* requires subexpressions), depend on training data or codebooks (*NeuReduce*, *SSPAM*), are nondeterministic (due to sampling of input data, e.g. *QSynth*, *Syntia*) or struggle with certain expressions (e.g., with nonlinear expressions or expressions with more than 5 variables).

While linear MBAs have been shown to be easily solvable in general, it is still an open question whether more generic MBAs can constitute a solid obfuscation technique. Although the papers describing *MBA-Solver* [26] and *MBA-Flatten* [13] claim to be capable of simplifying polynomial and even nonpolynomial MBAs, the published algorithms do not fully support this. They restrict the representation of input MBAs[1], and for a nonpolynomial input MBA it is required to provide a subexpression whose replacement by a variable makes the MBA a polynomial one, which is clearly impractical. Moreover, they do not output simplest solutions, but only ones using a set of bitwise base expressions.

1.2 Contribution

In this paper, we turn our attention to the simplification of **nonlinear** MBAs, using *SiMBA* as the core tool; we review preliminaries in Section 2. Although *SiMBA* is meant to operate on linear MBAs, it can in fact correctly solve all MBAs which are reducible to linear ones (see Section 3). We propose four improvements to *SiMBA* in Section 4 to make it more flexible. Then we describe a more involved algorithm called *GAMBA* (for *Generalized Advanced MBA* simplifier) for nonlinear MBAs (Section 5) based on an iterative application of *SiMBA*, in combination with additional tricks

[1] E.g., the user has to know whether an MBA is linear, polynomial or nonpolynomial; the number of variables is limited to 4; variables have to use prescribed variable names; a polynomial input MBA is required to be a sum of monomials; each term's first factor has to be a constant.

and, critically, a substitution of nonlinear subexpressions by variables. While we cannot claim to be able to solve all MBAs with the current implementation, evaluation with publicly available datasets shows very promising results (Section 6).

2 Preliminaries

2.1 Mixed Boolean-Arithmetic Expressions

Mixed Boolean-arithmetic expressions use logical (or bitwise, resp.) operators as well as arithmetic ones. While logical operators basically operate on $B = \{0, 1\}$, i.e., single bits, bitwise operations are equivalently applied to all bits of n-bit words in B^n for any fixed $n \in \mathbb{N}$. This connection is of great importance, as it builds the foundation for the generation as well as the algebraic simplification of linear MBAs [27].

As in [19], we prefer the notion of bitwise expressions which fits better to our context. That is, we use the operators & (bitwise conjunction), ^ (bitwise exclusive disjunction), | (bitwise inclusive disjunction) and ∼ (bitwise negation) rather than $\wedge$ (logical conjunction), $\oplus$ (logical exclusive disjunction), $\vee$ (logical inclusive disjunction) and $\neg$ (logical negation).

The most popular classes of MBAs are that of linear and polynomial ones, resp. We first reiterate the definition of a polynomial MBA as in [19].

Definition 1 *Let $B = \{0, 1\}$ and $n, t \in \mathbb{N}$. A polynomial mixed Boolean-arithmetic expression with values in B^n and t variables is a function $e : (B^n)^t \to B^n$ of the form*

$$e(x_1, \ldots, x_t) = \sum_{i \in I} a_i \prod_{j \in J_i} e_{ij}(x_1, \ldots, x_t),$$

where $I \subset \mathbb{N}$ and $J_i \subset \mathbb{N}$, for $i \in I$, are index sets, $a_i \in B^n$ are constants and e_{ij} are bitwise expressions of variables $x_1, \ldots, x_t \in B^n$ for $j \in J_i$ and $i \in I$.

As is easy to see and already noted in [19], a linear MBA is a special kind of a polynomial one:

Definition 2 *Let $B = \{0, 1\}$ and $n, t \in \mathbb{N}$. A linear mixed Boolean-arithmetic expression with values in B^n and t variables is a function $e : (B^n)^t \to B^n$ of the form*

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

where $I \subset \mathbb{N}$ is an index set, $a_i \in B^n$ are constants and e_i are bitwise expressions of $x_1, \ldots, x_t$ for $i \in I$.

For instance, the MBA $x + (x \& y) - 2(x|y) + 42$ is linear, while the MBA $y(x \wedge y) - (x \& y)^2 - 1$ is polynomial, but not linear.

These notions were first introduced by Zhou et al. [27]. In this paper, we additionally use the term *general MBA* for MBAs which are not necessarily polynomial. Here we restrict on those using the bitwise operations &, |, ^ and ∼ as well as additions, subtractions, multiplications and powers (and implicitly left shifts, which can be written as multiplications of powers of 2).

There are two possible reasons why such an MBA is nonpolynomial:

1. It incorporates powers with nonconstant MBAs in their exponents, e.g., $3x^y + x + 17$. It seems obvious to us to call it an *exponential* MBA in this case.

2. It contains nontrivial constants or arithmetic operations in bitwise operations, e.g., $5 + (x|3) - (5\&y)$. In this case we call it *mixed.*

See Figure 1 for a visualization. By "nontrivial" constants, we mean constants that are neither 0 nor -1 since those are the counterparts to the logical truth values.

In their paper [27], Zhou et al. describe a fundamental relation between Boolean and bitwise expressions that paves the way for a surprisingly simple, but popular method for generating linear MBAs:

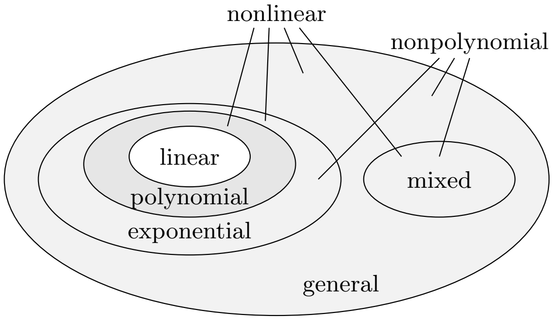


Figure 1: MBA types

Theorem 1 *Let $n, s, t \in \mathbb{N}$, x_i variables over B^n for $i = 1, \ldots, t$ and $e_j : (B^n)^t \to B^n$ bitwise expressions on these variables for $j = 1, \ldots, s$. Let*

$$e(x_1, \ldots, x_t) = \sum_{j=1}^{s} a_j e_j (x_1, \ldots, x_t)$$

be a linear combination of these bitwise expressions with coefficients $a_j \in B^n$ for $j = 1, \ldots, s$ and hence a linear MBA. Furthermore let, again for $j = 1, \ldots, s$, $\overline{e}_j : B^t \to B$ be the logical expression corresponding to e_j. Enumerate the possible combinations of zeros and ones for the variables by $B_t = \{b_1, \ldots, b_{2^t}\}$ arbitrarily, but fixed, and let $A = (v_{ij}) \in B^{2^t \times s}$ be the matrix of truth values of the $\overline{e}_j$'s with $v_{ij} = \overline{e}_j(b_i)$.

Then $e \equiv 0$ if and only if the vector $Y = Y_a = (a_1, \ldots, a_s)^T$ is a solution of the linear equation system $AY = o$ over B^n, where $o = (0, \ldots, 0)^T$ is the zero vector in B^{2^t}.

According to this theorem, a linear MBA e is equivalent to a bitwise expression $\tilde{e}$ on whole B^n, for any $n \in \mathbb{N}$, if and only if they are equivalent on B. As will be noted in Theorem 2, this also holds if $\tilde{e}$ is a linear MBA itself.

This builds the foundation for our simplifier *SiMBA* for linear MBAs and may also do so for generating general MBAs. To our knowledge, there is no obvious way how to define a similar approach for generating nonlinear MBAs via just an evaluation on inputs in $\{0, 1\}$. An example showing that these values are not unique for nonlinear MBAs in general is easy to find. E.g., the polynomial MBA x^2 is equivalent to the linear MBA x on $\{0, 1\}$.

2.2 Simplification of Linear Mixed Boolean-Arithmetic Expressions

Recent research has shown that linear MBAs are for sure those which are easiest to solve. This is why most existing MBA simplifiers are especially successful with these MBAs or even only allow them as inputs. From the mentioned algebraic tools, *MBA-Blast*, *MBA-Solver* and *SiMBA* are all based – more or less directly – on Theorem 1 while *MBA-Flatten* uses a different approach to transform an input expression into a linear combination of conjunctions.

One main difference between *SiMBA* on the one hand and *MBA-Blast* as well as *MBA-Solver* on the other hand is that the latter use Theorem 1 directly to transform each bitwise expression into a linear combination of bitwise expressions or a so-called *signature vector* and then combine the results, *SiMBA* uses slightly more involved insights that allow to evaluate a linear MBA as a whole on $B = \{0, 1\}$ and directly derive a first solution. We restate the underlying theorem [19]:

Theorem 2 *Let e and f be linear MBAs over words of the same length n and let $t \in \mathbb{N}$ be their (maximum) number of variables. Then $e \equiv f$ if and only if $e(b) = f(b)$ for each possible combination b of in total t zeros and ones.*

What all these tools have in common is that they derive a linear combination of base bitwise expressions as a candidate for a solution. One possible base is the set of all conjunctions, e.g.,

$\{x, y, x\&y, -1\}$ for two variables, but there are multiple candidates and, e.g., *MBA-Solver* uses a more complex basis. Therefore, the tools try to refine candidate solutions in order not to miss particularly simple solutions. While *MBA-Blast* and *MBA-Solver* check whether there is an equivalent expression using only one term, *SiMBA* guarantees to find the simplest solution for all inputs using two variables and for most inputs using three variables.

While the usage of *MBA-Blast* and *MBA-Solver* is restricted to MBAs with up to four variables due to their usage of lookup tables, with a number of entries which grows hyperexponentially following the formula 2^{2^t} for a number t of variables, this insufficiency is eliminated by *MBA-Flatten*, paying with a higher runtime and the lack of the possibility to find simplest solutions. In contrast, *SiMBA* is implemented fully generically and hence usable for an arbitrary number of variables, finding simple solutions for all inputs which are reducible to MBAs with at most three variables. We will explain in Section 4 how this can be made even more flexible.

3 Simplification of MBAs Reducible to Linear Ones Using *SiMBA*

Although *SiMBA* is mainly targeting linear MBAs, its usage is not restricted to those. Since it evaluates expressions directly on combinations of zeros and ones, its results are fully independent of the input expressions' representations. Unless *SiMBA* is instructed to neglect nonlinear inputs, it provides a result for any input MBA.

On the one hand, it is obvious that not every MBA is reducible to a linear one. But on the other hand, it is similarly easy to see that for each arbitrary MBA one can construct a linear one which corresponds to it for values in $\{0, 1\}$. Consequently, and as already motivated in Subsection 2.1, *SiMBA*'s results are incorrect if and only if the input MBA has no equivalent representation as a linear MBA.

As a consequence, it is meaningful to run *SiMBA* on (not necessarily linear) MBAs whenever it is known that they are reducible to linear ones, i.e., their ground truths are linear. Since it can be assumed that a large portion of MBAs in practice are generated from linear ones, this extends *SiMBA*'s field of application significantly.

Without any knowledge about the ground truth, one can still run the simplifier, but the result has to be verified. This can be done, e.g., via an evaluation of both expressions or their difference for values in B^n. However, this might take too long for larger n, and a check for just a subset of possible argument combinations will not give 100% certainty.

Note that, e.g., a polynomial MBA with t variables is in general not a polynomial in these variables. It is well a polynomial in bitwise expressions and can be transformed into a polynomial in at most 2^t base bitwise expressions, but when replacing the latter by variables, the information about their interdependence is lost. Hence, it is not obvious how basic analytic techniques can be leveraged to find roots of polynomial MBAs.

For example, consider the polynomial MBA

$$\begin{aligned} e_1 = &(-{\sim}(x|y) + (x|{\sim}y))(-(x \,\hat{}\, y) - {\sim}(x \,\hat{}\, y)) \\ &+ (-2{\sim}(y|x) + {\sim}x + {\sim}(y \,\hat{}\, x))(-{\sim}y - y) \end{aligned}$$

and the exponential MBA

$$\begin{aligned} e_2 = &(-{\sim}(x|y) + (x|{\sim}y))^{-(x \,\hat{}\, y) - {\sim}(x \,\hat{}\, y)} \\ &+ (-2{\sim}(y|x) + {\sim}x + {\sim}(y \,\hat{}\, x))^{-{\sim}y - y}, \end{aligned}$$

which can both be simplified to $x + y$.[2] For another polynomial MBA

$$e_3 = (x\&y)(x|y) + (x\&{\sim}y)({\sim}x\&y),$$

[2] The second factor of each term in e_1 vanishes when writing negations ${\sim}X$ as $-X - 1$, and equivalently for the exponents in e_2.

SiMBA would output $\tilde{e}_3 = x\&y$ as a result. It is easy to verify that $\tilde{e}_3$ is equivalent to e_3 on $\{0,1\}$, but we have, e.g., that $e_3(1,2) = 2$, while $\tilde{e}_3(1,2) = 0$. In fact, it is not possible to transform e_3 into a linear MBA.

4 Extending the Flexibilty of *SiMBA*

We want to use *SiMBA* as a utility for the simplification of general MBAs (Section 5). For this purpose, we perform the following adaptations.

4.1 Complexity Metrics

SiMBA finds for most expressions with two or three variables a solution with a minimal number of terms [19]. Obviously, such a solution is not necessarily the intuitively simplest solution. For instance, the expressions $e_4 = 2((y\&{\sim}z)|(x\&(y|{\sim}z))) - (x \,^\wedge\, y \,^\wedge\, z)$ and $\tilde{e}_4 = x + y - z$ are equivalent. While the former has fewer terms, the latter is intuitively the one which would be expected as a simplification result.

In general, the decision which solution is the "simplest" may depend on the type of application as well as the user's perspective. Hence, we extend *SiMBA* to choose a metric to guide decisions. Some possible metrics are based on the representation of an expression as an AST, which we prefer over a *directed acyclic graph (DAG)*, in which equivalent nodes are merged. DAG nodes are impractical to us since coincident copies of a subexpression might be simplified in different ways. We use the following metrics:

1. **MBA alternation:** Proposed by Eyrolles [5] based on a DAG representation, it measures how often an expression alternates between bitwise and arithmetic operations. Thus, a purely bitwise or purely arithmetic expression has an MBA alternation of zero, independently of its number of terms.

2. **Number of nodes in the AST representation:** Here we simply count the AST nodes, similarly to the DAG nodes, as suggested by Eyrolles [5].

3. **Number of terms:** This is the metric which we have used so far and which is easiest to compute and optimize for in our case.

4. **String length:** The string length is easy to compute too, but does not give much insights into the structure of an MBA. Besides, it depends on the format of an MBA's string representation.

If two considered MBAs have the same value for a metric, we apply a secondary metric to make a decision. Given a vector of results of an MBA for all possible combinations of arguments in $\{0,1\}$, it is easiest for us to determine the minimal number of terms we can achieve, while the other metrics' optimizers are harder to find. This extension implies that we have to compute more possible solutions and compare them.

In order to keep the number of inspected solutions small, we do not consider any solution with an equal or higher number of terms compared to the linear combination of conjunctions, i.e., the first candidate solution. We do not expect any such solution to minimize any of the metrics above, as the bitwise expressions in this linear combination are very simple.

Returning to the example above, the number of terms is the only metric which would prioritize e_4 over $\tilde{e}_4$; no other possible solution is additionally investigated.

Digging a bit deeper, the candidate solution $\tilde{e}_4$ is the initially determined linear combination of conjunctions, while e_4 is found in an attempt to decompose the input MBA's result vector $(0,1,1,2,-1,0,0,1)^\tau$ into a linear combination of truth value vectors of at most two bitwise expressions:

$$2\,(0,1,1,1,0,0,0,1)^\tau - (0,1,1,0,1,0,0,1)^\tau.$$

If we would consider all possible solutions consisting of three terms too, we would have to handle a surprisingly high number of different solutions: Even when fixing the bitwise expressions' coefficients to 1, 1 and -1, we can derive $3^5 = 243$ possible solutions, all using a different selection of three terms. In fact, even the set of feasible coefficients is only bounded by the number of bits, since, amongst other choices, $2, a$ and $-a-1$ would yield a positive number of solutions for any a.

4.2 Additional Refinement Attempts

SiMBA, as presented in [19], performs up to 8 attempts to decompose a result vector in order to find a simple solution. We already indicated in the preceding subsection that we now perform in some occasions an exhaustive search for suitable bitwise expressions and coefficients that yield the same number of terms.

However, the algorithm still would not find solutions such as $x + {\sim}y$ or ${\sim}x + {\sim}y$. It is important to find such simple solutions in order to have an optimal chance to simplify complex general MBAs in the sequel. So we extend *SiMBA* by approaches to find linear combinations of a bitwise expression and a negated one, as well as linear combinations of two negated bitwise expressions.

Let e be a linear MBA using $t \geq 2$ variables and $F = (e(b_1), \dots, e(b_{2^t}))^\tau$ its vector of results when evaluated for all possible combinations b_i of zeros and ones, where b_i assigns the value 1 to the variable x_j if i's remainder after a division by 2^j is larger than 2^{j-1}. We extend *SiMBA* with the following two approaches to reduce the results' complexity:

1. If F has three or four distinct values, its first entry a is nonzero and there are at most two values that are neither a nor $2a$, we can express e as a linear combination of an unnegated and a negated bitwise expression in the following cases:

   (a) If there is one such value b, we have two possible linear combinations with either coefficient $b - a$ or $b - 2a$ for the unnegated bitwise expression and coefficient $-a$ for the negated one.

   Example for $t = 2$: The vector $(2, 2, 1, 4)^\tau$ can be decomposed into

   $$(0, 0, -1, 0)^\tau - 2\,(-1, -1, -1, -2)^\tau,$$

   yielding the solution $-({\sim}x\&y) - 2 \cdot {\sim}(x\&y)$, or

   $$(0, 0, -3, 0)^\tau - 2\,(-1, -1, -2, -2)^\tau,$$

   yielding the solution $-3({\sim}x\&y) - 2 \cdot {\sim}y$.

   (b) If there are two such values b, c and, w.l.o.g., $c - b = a$, we have a linear combination with coefficient $b - a$ for the unnegated bitwise expression and coefficient $-a$ for the negated one.
   Note that in this case we might have multiple possible decompositions if $b = 0$, since we can express F's entries equal to $-a$ alternatively as $b - a$.

   Example for $t = 2$: The vector $(-1, 0, 1, 0)^\tau$ can be decomposed into

   $$2\,(0, 1, 1, 1)^\tau + \,(-1, -2, -1, -2)^\tau,$$

   yielding the solution $2\,(x|y) + {\sim}x$.

2. If F has three or four distinct values, its first entry a is nonzero, there are exactly two values b, c that are neither a nor $2a$ and these sum up to $3a$, we can express e as a linear combination of negated bitwise expressions with coefficients $a - b$ and $a - c$.

   The corresponding negated bitwise expressions' result vectors have value -1 where F has either value a or c (or b, resp.) and value -2 where F has either value $2a$ or b (or c, resp.). The truth values of the bitwise expressions to be negated can then be derived by replacing -1 by 0 and -2 by 1.

Example for $t = 2$: The vector $(4, 9, 9, 3)^\tau$ can be decomposed into

$$(-1, -1, -1, -2)^\tau - 5\,(-1, -2, -2, -1)^\tau,$$

yielding the solution $\sim(x\&y) - 5 \cdot \sim(x \wedge y)$.

The attentive reader might have noticed that by ignoring values equal to $2a$ we have neglected cases in which we can find solutions with just one term, as these are already found in a previous step of the algorithm – see Subsection 3.2.4 of [19].

4.3 Try to Split Expressions with More Variables

If the initially determined linear combination of conjunctions uses more than three variables, this indicates that the input expression cannot be expressed using a fewer number of variables, and we thereby cannot use a lookup table for finding a simpler solution. Note that a lookup table for four variables would have $2^{2^4} = 65\,536$ entries, and one for five variables would already have $2^{2^5} = 4\,294\,967\,296$ entries.

In the next subsection we provide one possible workaround, but this increases the runtime for a higher number of variables. Yet we can avoid it in special cases: If we can nontrivially partition the initial solution's terms with respect to the occurring variables, i.e., such that the parts use disjunct sets of variables, we can handle these parts separately and combine the results. An optional constant may fit to either part, so we may have multiple possible solutions.

Consequently, for parts using at most three variables, we have the chance to run the usual procedure. For others we can apply the method described in the next subsection. As an example, consider the input expression

$$\begin{aligned}&(a\&{\sim}b) + b - 3((x\&{\sim}y) \wedge z) + 3({\sim}y|z)\\&\quad - ((x\&{\sim}y) \wedge {\sim}z) + 4({\sim}x|y) - 4({\sim}x \wedge (y\&z))\\&\quad + (x \wedge (y\&{\sim}z)) - x - 2({\sim}x\&(y|{\sim}z))\\&\quad - 2((x\&y) \wedge z),\end{aligned}$$

which can be transformed into the initial solution

$$\begin{aligned}&a + b - (a\&b) - 2y - 2z + 2(x\&y) + 2(x\&z)\\&\quad + 4(y\&z) - 4(x\&y\&z).\end{aligned}$$

This can be partitioned into three terms using the variables a and b and six terms using the variables x, y and z. While the former can be simplified to $a|b$, the latter reduces to $-2({\sim}x\&(y \wedge z))$. Hence, the result would be

$$(a|b) - 2({\sim}x\&(y \wedge z)).$$

If a constant term exists in the initially determined linear combination, it fits to any partition. However, it is reasonable to choose the option which minimizes the used complexity metric.

4.4 Creation of Base Bitwise Expressions

As mentioned, lookup tables of bitwise expressions grow fast with an increasing number of variables, hence we only use them for up to three variables. The peer tools *MBA-Blast* and *MBA-Solver* do so also for four variables, but as noted in [19], it slows the algorithms down drastically.

Fortunately, we can instantly create sufficiently simple bitwise expressions for any given vector of truth values using the *Quine–McCluskey algorithm* [14,18] to find a minimal disjunctive normal form, i.e., a disjunction of conjunctions of (possibly negated) variables that cannot be further reduced. Note that unfortunately its exponential runtime limits its usage too.

As an example, consider the truth value vector $(0,1,1,0,0,1,1,0)^\tau$ for four variables. The Quine-McCluskey algorithm would first create conjunctions for each 1 in this vector, namely $x\&{\sim}y\&{\sim}z$, $x\&{\sim}y\&z$, ${\sim}x\&y\&{\sim}z$ and ${\sim}x\&y\&z$, and then merge the former two as well as the latter two to finally get the disjunction $(x\&{\sim}y)|({\sim}x\&y)$. Actually this can be further simplified to $x \wedge y$.

In order to identify chances to find simpler bitwise expressions as above, we perform refinements iteratively until nothing changes any more:

1. *Try to insert exclusive disjunctions:* For any bitwise subexpressions X and Y, we can use the following patterns for a transformation into an exclusive disjunction $X \wedge Y$:
$$(X\&{\sim}Y)|({\sim}X\&Y) \equiv X \wedge Y,$$
$$(X|Y)\&({\sim}X|{\sim}Y) \equiv X \wedge Y.$$

2. *Potentially flip negations:* If a subexpression becomes simpler via flipping all its operands' negations, apply *De Morgan's law.* Additionally, ${\sim}X \wedge {\sim}Y \equiv X \wedge Y$.

3. *Try to factor out common subexpressions:* If a certain subexpression occurs in all operands of another subexpression, we apply the *distributive law.* This includes the following well-known patterns:
$$(X\&Y)|(X\&Z) \equiv X\&(Y|Z),$$
$$(X|Y)\&(X|Z) \equiv X|(Y\&Z).$$

This list is not exhaustive and may be extended as desired. Note that other common rules such as the *absorption laws* or the *idempotence laws* have already implicitly been applied during the Quine-McCluskey algorithm.

Algorithm 1 Simplification of an MBA e reducible to a linear one (extended *SiMBA*)

```
1. Determine linear combination $\tilde{e} \equiv e$ of conjunctions
2. Identify the number $t$ of variables in $\tilde{e}$
3. If $t \leq 3$:
    (a) Try to find a simpler solution using table
4. Else:
    (a) Determine partition $P$ of $\tilde{e}$ w.r.t. variables
    (b) For all parts $p \in P$ with at most 3 variables:
        i. Try to find a simpler solution using table
    (c) For all parts $p \in P$ with more than 3 variables:
        i. Try to find a simpler solution using bitwise creation as described in Subsection 4.4
    (d) Compose the results
```

Algorithm 1 summarizes *SiMBA*'s steps for simplifying MBAs that are reducible to linear ones. Here the whole branch 4 is newly introduced, while step 3a has been extended.

5 Simplification of General MBAs

In this section, we describe how we can use *SiMBA* in combination with additional steps to simplify MBAs that are not necessarily linear and not necessarily reducible to linear MBAs. As *SiMBA*, *GAMBA* is written in `Python` without usage of packages such as `NumPy` or `SymPy` for nontrivial computations or simplification.

5.1 Outline

It is meaningful to operate on *abstract syntax trees (ASTs)* rather than on strings. This helps identify variables, classify nodes as bitwise, linear or nonlinear subexpressions, compare expressions with more flexibility (e.g., not necessarily requiring a coincident order of operands), as well as extract, modify and reintegrate subexpressions.

Each node of an AST is either a variable, a constant or an operator with a certain number of operands. That is, each leaf node is a variable or a constant. The following operators, ordered by precedence, suffice for our purposes: power, bitwise negation ($\sim$), multiplication ($\cdot$), sum ($+$), conjunction ($\&$), exclusive disjunction (^) and inclusive disjunction ($|$).

The strategy of *GAMBA* is sketched in Algorithm 2 which iteratively identifies linear subexpressions and simplifies them using *SiMBA*. In order to increase the chance of doing so, between these simplification runs, operations that support simplification as well as normalization are performed.

Algorithm 2 Simplification of a general MBA e (*GAMBA*)

```
1. Parse $e$ into an AST $t$
2. Repeat until convergence:
   (a) Refine $t$ as described in Subsection 5.2
   (b) Identify linear subtrees of $t$ as described in Subsection 5.3
   (c) Try to factorize nonlinear sums as described in Subsection 5.4
   (d) Apply SiMBA to linear subexpressions as sketched in Algorithm 1
   (e) Collect nodes for substitution
   (f) For all combinations of those nodes:
       i. Substitute these nodes by variables
       ii. Apply steps 2a to 2d
       iii. Back-replace these variables
       iv. Refine $t$
3. Polish $t$ for optimal representation
4. Return a string representation of $t$
```

In some cases it is nontrivial to retrieve linear parts of a nonlinear MBA if, e.g., products of linear subexpressions are expanded. To resolve that, *GAMBA* incorporates a factorization procedure which tries to decompose sums of higher-order terms into factors.

Another main challenge for the simplification of MBAs that contain constants or arithmetic expressions within bitwise operations is to get rid of the latter or any other nonlinear subexpressions, if possible. This is done via a substitution logic, making subexpressions linear by substitution of nonlinear parts with temporary variables, simplifying and reinserting the substituted parts.

The aim of step 3 is to apply some standardization, including a deterministic order of operands in any kind of operation. This facilitates comparisons between simplified expressions.

This algorithm is a proof-of-concept and may have to be further adapted in order to be powerful enough to simplify (nearly) arbitrarily complex expressions.

5.2 Node Operations for Refinement

A refinement procedure is very crucial in order to establish invariants as well as to make sure that we can leverage a successful simplification of a subexpression. Imagine, e.g., that some expression cannot be further simplified because it has a subexpression like $(x|3)\&{\sim}(x|3)$, which is equivalent to 0 and hence purely bitwise.

For normalization, we establish some trivial invariants such that all constant operands of a node are merged into one. Additionally we perform, amongst others, the following inspections in order to prepare the expression for simplification:

Applying logical rules in bitwise operations: In conjunctions and exclusive as well as inclusive disjunctions, we get rid of all constants that are 0 or -1, and resolve duplicate operands as well as operands that are inverse to each other. Furthermore *De Morgan's law* may be applied to conjunctions or inclusive disjunctions.

Rearrangement of sums: In sums, we collect terms that differ at most in constant factors and factor out common factors if possible.

Merging of powers: In multiplications, powers are merged if they have the same base. This helps identify linear MBAs in exponents.[3]

Eliminating or rewriting bitwise negations: Nested negations are resolved, respecting the possibility that $\sim X$ is written as $-1 - X$ or $-1(1 + X)$ for any subexpression X. Furthermore, for any bitwise negation, either written explicitly using the negation operator or implicitly in arithmetic form, it is decided upon its context which representation increases the chance of simplification.

Flattening bitwise operations: In some occasions a bitwise operation's complexity can be reduced via splitting it in terms, which is mainly meaningful in sums. This is, e.g., possible if an inclusive disjunction's operands are (or can be made) disjunct, including patterns such as $(X\&Y)|(X \wedge Y) \equiv (X\&Y) + (X \wedge Y)$ or even $X|(X \wedge Y) \equiv (X\&Y) + (X \wedge Y)$, which is equivalent to the former.[4]

Factoring out from bitwise operations: Powers of 2 can be factored out from conjunctions, inclusive and exclusive disjunctions if they appear in all their operands. This can help get rid of constants in bitwise operations and hence make subexpressions linear. This corresponds to the pattern $2X\&2Y \equiv 2\,(X\&Y)$, and equivalent for the other operations. In fact we can even factor out a power of 2 in cases where not all operands are divisible by that. Depending on the type of operation, we may have to compensate this by adding a remainder. Consider the following examples for subexpressions X and Y and a constant a and remember that $\sim X \equiv -X - 1$:

$$\begin{aligned}
\sim(2X)\&2Y &\equiv 2\,(\sim X\&Y),\\
\sim(2X)|2Y &\equiv 2\,(\sim X|Y) + 1,\\
(2a+1) \wedge 2X &\equiv 2\,(a \wedge X) + 1.
\end{aligned}$$

Merging bitwise operations involving constants: In sums, we may get rid of constants via merging bitwise operations with constants and, apart from that, coincident operators. This is particularly useful if the arising constants are 0 or -1. The following rules hold for constants a, b that have no 1s in common in their binary representations:

$$\begin{aligned}
(a\&X) + (b\&X) &\equiv (a+b)\&X,\\
(a|X) + (b|X) &\equiv ((a+b)|X) + X,\\
(a \wedge X) + (b \wedge X) &\equiv 2\,(\sim(a+b)\&X) + a + b,\\
(a|X) - (b\&X) &\equiv (\sim(a+b)\&X) + a,\\
(a \wedge X) - 2\,(b\&X) &\equiv 2\,(\sim(a+b)\&X) - X + a,\\
(a \wedge X) + 2\,(b|X) &\equiv 2\,(\sim(a+b)\&X) + X + a + 2\,b.
\end{aligned}$$

Merging bitwise operations involving inverse elements: Similarly as explained above, terms of sums may be merged if the disjunct constants are replaced by inverse elements which are disjunct too.

[3] As an exponential MBA can be generated from a polynomial one by adding linear MBAs that are equivalent to 1 as exponents and optionally splitting the arising powers, we try to restore such exponents.

[4] Since a bit of $X \wedge Y$ is 1 if the corresponding bits in X and Y are different, it suffices to assume that they coincide in the inclusive disjunction's other operand, implying we can add a conjunction with Y.

Some of the resulting patterns are well known:

$$\begin{aligned}(X\&Y)+(\sim X\&Y)&\equiv Y,\\(X|Y)+(\sim X|Y)&\equiv -1+Y,\\(X \mathbin{^\wedge} Y)+(\sim X \mathbin{^\wedge} Y)&\equiv -1,\\(X|Y)-(\sim X\&Y)&\equiv X,\\(X \mathbin{^\wedge} Y)-2\,(\sim X\&Y)&\equiv X-Y,\\(X \mathbin{^\wedge} Y)+2\,(\sim X|Y)&\equiv \sim X+Y-1.\end{aligned}$$

Note that we do not have to consider any optimization of linear expressions since *SiMBA* already outputs simplest expressions for them. However, the shown patterns also hold for non-bitwise subexpressions X and Y.

5.3 Identification of Linear Subexpressions

Before *SiMBA* can be applied to linear subexpressions, it is necessary to identify them. Additionally, purely bitwise expressions are identified as well. This is done in a straightforward way by induction:

- A variable is a bitwise expression.
- A constant node is considered bitwise if it is 0 or -1 (corresponding to the logical constants), and linear otherwise.
- A subexpression corresponding to a bitwise operator (negation, conjunction, exclusive or inclusive disjunction) node is bitwise if all its operands are bitwise expressions, and nonlinear otherwise.
- A sum is nonlinear if it has a nonlinear term, and linear otherwise. In the former case, we can collect its linear terms which consequently form a linear subexpression.
- Being able to assume that at most one operand is constant, a product is nonlinear if it has more than two operands, at least one nonlinear operand or two operands and none of them is constant. Otherwise it is linear.
- Being able to assume that a power is never trivial, i.e., does not have the constant 1 as exponent, it cannot be linear.

5.4 Factorization

The factorization of nonlinear sums is a crucial step for simplifying nonlinear MBAs whose linear parts are well-hidden by an expansion of their products. In such cases we cannot identify linear subexpressions easily, and the basic node operations as described in Subsection 5.2 do not provide a solution. As an example, consider the MBA

$$\begin{aligned}-x\cdot{\sim}(x|z)-y\cdot{\sim}(x|z)-x\,(x\&{\sim}z)&\\-y\,(x\&{\sim}z)-xz-yz,&\end{aligned}$$

which is in fact the product of the linear MBAs $x+y$ and

$$-({\sim}(x|z))-(x\&{\sim}z)-z,$$

whereas the latter is equivalent to a constant 1.[5] That is, in order to simplify the MBA to $x+y$, we have to identify these factors. *GAMBA* iteratively finds simple factors which appear in a large

[5] This becomes evident after applying De Morgan's law to ${\sim}(x|z)$ to obtain ${\sim}x\&{\sim}z$, realizing that $({\sim}x\&{\sim}z)+(x\&{\sim}z)$ is equivalent to ${\sim}z$ and writing the latter as $-z-1$.

number of terms of a sum, factors them out of them and splits the sum accordingly. In the sequel, these new terms can be collected if they only differ in the components that have been factored out.

In the previous example, x and y both appear in three terms. Hence, they would be factored out to obtain

$$\begin{aligned} &x\,(-({\sim}(x|z)) - (x\&{\sim}z) - z)\\ &\qquad + y\,(-({\sim}(x|z)) - (x\&{\sim}z) - z), \end{aligned}$$

these terms can be combined to

$$(x+y)\,(-({\sim}(x|z)) - (x\&{\sim}z) - z),$$

and the latter factor vanishes by simplifying using *SiMBA*.

Before factorization, it may be necessary to expand products and powers and collect terms thereafter. Note that an MBA generator may, after generating linear MBAs for subexpressions, obscure those via expanding products and factorizing expressions into factors that cannot be simplified easily.

5.5 Substitution of Subexpressions

The techniques presented so far are usually not sufficient to simplify nonlinear MBAs, especially mixed ones, i.e., those containing constants or arithmetic operations within bitwise operations. In order to get rid of those, we apply some substitution logic. Our hope is to transform nonlinear subexpressions into linear ones via substitution of parts by variables, and that they get simpler via simplification using *SiMBA* and subsequent reinsertion of the substituted parts. As a simple example, consider the MBA

$$((-x)\,\hat{}\,y) - 2\,(({\sim}-x)\&y)$$

which has no nontrivial linear subexpressions, but is in fact easily solvable by *SiMBA* after substituting $-x$ by a temporary variable, say, X. Then the expression

$$(X\,\hat{}\,y) - 2(({\sim}X)\&y)$$

would resolve to $X-y$,[6] and after resubstitution to $-x-y$.

In some cases it might not be sufficient to only replace one subexpression because, e.g., subexpressions might remain nonlinear, but substituting a second subexpression might help. For instance, this is the case for the MBA

$$\begin{aligned} &{\sim}x + {\sim}(y-1) + 2\\ &\qquad + ((-{\sim}x + 1 - 1)|(-({\sim}(y-1)) - 1)), \end{aligned}$$

which can only be simplified after a simultaneous substitution of $-({\sim}x+1)-1$ by, say, X, and $-{\sim}(y-1)-1$ by, say, Y.

In general, it is hard to decide on the right strategy since a simultaneous substitution may in some occasions hide too much of the interdependence between variables and subexpressions, so we collect all nodes which are meant to be substituted and run the substitution procedure on all possible subsets of this set.

In fact, the example above shows that we have to identify subexpressions to be substituted even if they are not fully present: We actually substitute ${\sim}x+1$ by $-X-1$ and ${\sim}(y-1)$ by $-Y-1$. This simplifies to ${\sim}(X\&Y)$ and to $-x|-y$ after back-substitution and refinement.

After substitution it might be necessary to do additional work regarding linear subexpressions which is usually done by *SiMBA*, but not in this case, when the interdependence between subexpressions is hidden by the substitution. In this course, e.g., we check whether terms of sums cancel out due to basic laws of logic.

[6] This can be seen after transforming $X\,\hat{}\,y$ into ${\sim}(({\sim}X)\,\hat{}\,y)$ and consequently into $-(({\sim}X)\,\hat{}\,y)-1$, expanding $({\sim}X)\,\hat{}\,y$ to $(({\sim}X)|y) - (({\sim}X)\&y)$, replacing $({\sim}X|y) + (({\sim}X)\&y)$ by ${\sim}X+y$ and finally writing ${\sim}X$ as $-X-1$.

5.6 Remarks and Outlook

We have described and implemented various techniques that support the simplification of MBAs of any kind, and the experiments below will suggest that this is a sophisticated starting point on the way to a powerful MBA simplifier. However, simplifying nonlinear MBAs is far from being straightforward and hence we can neither guarantee nor hope that this algorithm will be able to simplify all possible inputs. We consider it work in progress which can be improved with every input it fails to simplify. There will be further techniques to apply or transformation rules that can be implemented.

Apart from that, the current state of *GAMBA* is not optimized regarding runtime. It may in some occasions consider subexpressions which are actually already optimally simplified. Moreover, we are aware of corner cases in which a tradeoff between performance and success probability has to be met, e.g., when a high number of possible solutions exist (see Subsection 4.1) or the number of variables is large (see Subsection 4.3).

Besides, a high number of different nonlinear subexpressions, including constants, within larger subexpressions is challenging if they cannot be resolved applying transformation rules. In our substitution logic we have to substitute them by different variables, which increases the runtime. Our main goal is to show that it is well possible to simplify MBAs of any kind, and suggest one possible way how to achieve that.

6 Experimental Results

All experiments are run on a Linux Mint 21 virtual machine on a single core of an Intel Core i7-12700K CPU at 3.6 GHz. The runtime was measured using Python 3.11 with the `time` package. Furthermore, we use $n = 64$ bits in all experiments unless otherwise noted.

We use six publicly available datasets. Table 1 shows their numbers of expressions, expression types (linear, polynomial, nonpolynomial), numbers of variables (Vars), the average numbers of MBA alternations (Alt ∅), and the AST node count averages (Node ∅). All occurring nonpolynomial expressions are mixed, i.e., have constants or arithmetic operations within bitwise operations.

Table 1: Public MBA datasets

Dataset	Expr.	Type	Vars	Alt ∅	Node ∅
NeuReduce [8]	10 000	linear	2 to 5	7.9	53.6
MBA-Obf. [11] linear	1 000	linear	2 to 3	30.6	267.7
MBA-Obf. [11] nonlinear	500	poly	2	18.7	94.8
	500	nonpoly	2	26.1	110.3
Syntia [13]	182	linear	1 to 2	1.6	9.4
	51	poly	1 to 3	3.6	17.0
	267	nonpoly	1 to 3	6.9	27.6
MBA-Solver [26]	1 000	linear	1 to 4	9.1	71.6
	1 000	poly	1 to 3	9.5	58.7
	1 000	nonpoly	1 to 3	57.4	306.1
QSynth EA [4]	500	nonpoly	1 to 3	77.6	281.7

Per dataset, we compute how many expressions can be simplified with *SiMBA* and *GAMBA*, and compare with peer tools. Timeout is 60 min unless noted otherwise and taken from references. Results reported are taken from the respective publications unless the tool name is written in *italic* letters. Then they are self-generated. We do so for *MBA-Flatten* since it is the closest and most recent related tool.

When presenting simplification results, $\equiv$ denotes that a complex MBA expression e has been reduced to the corresponding ground truth $e^\star$. Moreover, $\approx$ indicates that the resulting expression is semantically equivalent to $e^\star$ (e.g. by simplifying their difference or proving equivalence using a SMT solver such as Z3 [21]). We use $\times$ when the result could not be proven to be correct, no result was given, or an error occurred. The column % indicates the percentage of successful experiments ($\equiv, \approx$) over the dataset. Bold letters mark the best results per dataset.

Denote the simplification function by S. Given $(e,e^\star)$ in the benchmarking scenario, a simplified expression $S(e)$ can be verified in several ways:

i. It is identical to the ground truth, i.e., $S(e) \equiv e^\star$.

ii. It is identical to the simplified ground truth, i.e., $S(e) \equiv S(e^\star)$.

iii. Their simplified difference $S(S(e) - S(e^\star))$ is zero, hence the result is semantically equivalent.

iv. It can be proven semantically equivalent using an SMT solver. For runtime performance reasons, the last check is often only computed for words B^n with a limited number of bits, e.g., $n = 4$ or $n = 8$, allowing some incorrect results go undetected.

We put the focus on simplification success rather than runtimes. While we mention *SiMBA*'s and *GAMBA*'s runtimes in order to be able to understand the complexity of, e.g., the additional refinement logic as well as iterative calls to *SiMBA*, we refer to [12] and [19] for runtime comparisons among nonalgebraic and algebraic tools, resp.

6.1 NeuReduce

The test dataset of *NeuReduce* [8] consists of 10 000 linear expressions with 2 to 5 variables. We observe that *SiMBA* and *GAMBA* can simplify all expressions, even to the ground truth in all cases, see Table 2. Note that *MBA-Flatten* fails to handle expressions with 5 variables.

Table 2: NeuReduce dataset results [8], timeout 40 min

Tool	$\equiv$	$\approx$	$\times$	Timeout	%
Arybo	862	0	0	9 138	8.6
SSPAM	1 420	0	0	8 580	14.2
Syntia	842	734	8 424	0	15.8
NeuReduce	7 796	28	2176	0	78.2
MBA-Flatten	8 560	0	1 440	0	85.6
SiMBA	10 000	0	0	0	**100.0**
GAMBA	10 000	0	0	0	**100.0**

The expressions of this dataset are simplified by *SiMBA* in 0.77 ms (median: 0.74 ms) and by *GAMBA* in 8.30 ms (median: 9.27 ms) on the average.

6.2 MBA-Obfuscator

The dataset consists of 1 500 linear, 1 500 polynomial and 1 500 nonpolynomial MBA expressions with 2 to 4 variables, but results are reported in [11] only for the first 1 000 linear, and 1 000 nonlinear expressions (combining the first 500 polynomial and nonpolynomial expressions).

MBA-Obfuscator [11] was the first proposal to generate diversified nonlinear MBA expressions. Due to the construction procedure, the nonlinear expressions share the same linear ground truth

Table 3: MBA-Obfuscator linear dataset results [11]

Tool	$\equiv$	$\approx$	$\times$	Timeout	%
Arybo	N/A	569	0	431	56.9
SSPAM	N/A	386	356	258	38.6
Syntia	N/A	97	903	0	9.7
NeuReduce	N/A	756	244	0	75.6
MBA-Blast	N/A	1 000	0	0	**100.0**
MBA-Flatten	1 000	0	0	0	**100.0**
SiMBA	1 000	0	0	0	**100.0**
GAMBA	1 000	0	0	0	**100.0**

with the linear expressions. The linear expressions in the dataset are quite large: They comprise 267.7 AST nodes on average, with a mean MBA alternation count of 30.6 on average.

The linear subset does not pose any problem to *MBA-Blast*, *MBA-Flatten*, *SiMBA* or *GAMBA*, see Table 3. The nonlinear subset is much harder: *NeuReduce* can provide only one solution to an expression which is in fact still linear. *MBA-Flatten* fails on several nonpolynomial MBAs in the dataset. Surprisingly, *SiMBA* succeeds in simplifying all expressions thanks to their linear ground truth, as explained in Section 3. *GAMBA* in one case returns a slightly more complicated expression, but can still verify the result using verification strategy iii.

SiMBA runs 2.47 ms (median: 0.72 ms) on the linear and 0.60 ms (median: 0.62 ms) on the nonlinear dataset on the average, while *GAMBA* takes 18.41 ms (median: 10.14 ms) on the linear and 26.50 ms (median: 15.12 ms) on the nonlinear dataset on the average.

Table 4: MBA-Obfuscator nonlinear dataset results [11]

Tool	$\equiv$	$\approx$	$\times$	Timeout	%
Arybo	N/A	84	0	916	8.4
SSPAM	N/A	103	192	705	10.3
Syntia	N/A	98	902	0	9.8
Neureduce	N/A	1	999	0	0.1
MBA-Blast	N/A	147	853	0	14.7
MBA-Flatten	953	0	47	0	95.3
SiMBA	1 000	0	0	0	**100.0**
GAMBA	999	1	0	0	**100.0**

6.3 Syntia

This dataset contains 500 expressions with up to 3 variables and was generated using the Tigress obfuscator [3]. Although several expressions are duplicates (only differ by variable name, 438 expressions are unique), the dataset is used as a point of reference by many publications, e.g. [4], [13].

More than half of the expressions are nonpolynomial and 183 of them are not reducible to a linear MBA. Consequently *SiMBA* cannot simplify the latter, see Table 5. While *QSynth*, *MBA-Flatten* and *GAMBA* can simplify the entire dataset, only *GAMBA*'s results are always identical to the ground truth. Note that for solving the *Syntia* dataset, *MBA-Flatten* uses a non-generic, customized implementation based on knowledge about these MBAs' structure.

SiMBA's average runtime for this dataset is 0.18 ms (median: 0.10 ms) and that of *GAMBA* is 8.89 ms (median: 7.67 ms).

Table 5: Syntia dataset results [13]

Tool	≡	≈	×	Timeout	%
SSPAM	N/A	332	168	0	66.4
Syntia	N/A	369	131	0	73.8
QSynth	N/A	500	0	0	**100.0**
MBA-Blast	N/A	416	0	84	83.2
MBA-Solver	N/A	454	0	46	90.8
MBA-Flatten	302	198	0	0	**100.0**
SiMBA	317	0	183	0	63.4
GAMBA	500	0	0	0	**100.0**

6.4 MBA-Solver

The dataset contains 1 000 linear, 1 000 polynomial and 1 000 nonpolynomial MBA expressions with up to 4 variables. The nonpolynomial expressions are comparatively large: 57.4 MBA alternations and 306.1 AST nodes on average.

Table 6: MBA-Solver dataset results [13]

Tool	≡	≈	×	Timeout	%
SSPAM	N/A	705	320	1 975	34.2
Syntia	N/A	437	2 563	0	14.6
MBA-Blast	N/A	1 763	0	1 237	58.8
MBA-Solver	N/A	2 899	0	101	96.6
MBA-Flatten	2 500	443	0	57	98.1
SiMBA	1 757	87	1 156	0	61.5
GAMBA	2 998	2	0	0	**100.0**

Simplification results are shown in Table 6. *MBA-Solver* manages to solve 2 899 expression, but depends on subexpressions as additional input to steer the simplification process. *MBA-Flatten* produces 2 500 results equivalent to the ground truth and can additionally provide 443 semantically equivalent solutions. *SiMBA* can return results not only for linear expressions, but also for some nonpolynomial MBAs; yet it cannot solve any of the polynomial MBA expressions, as they are not reducible to linear ones. *GAMBA* can simplify all expression in the dataset, to an expression equivalent to the ground truth in almost all cases.

While *SiMBA*'s average runtime for this dataset is 10.39 ms (median: 1.65 ms), that of *GAMBA* is 19.12 ms (median: 11.96 ms).

6.5 QSynth EA

This dataset contains 500 nonpolynomial MBA expressions with up to 3 variables. The expressions were generated with the *Tigress* obfuscator [3] by applying its *EncodeArithmetic* (EA) transform. The expressions are large: 281.7 AST nodes on average. Furthermore, this dataset's average number of MBA alternations is 77.6, the highest in this comparison.

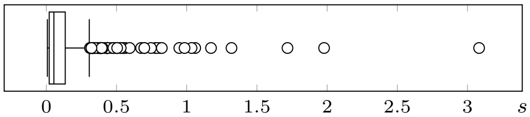
Table 7 shows that *GAMBA* can simplify 98.4% of these complex MBAs. For 61 expressions, the equivalence to the corresponding ground truths can be verified. It is worth to note that for the 8 runs where the verification was not successful (using verification methods iii or iv), in fact, the expressions were drastically shorter as well. *SiMBA* can only simplify 45 expressions, as the remaining ones have no linear ground truth. *MBA-Flatten* is not included here since it cannot handle the occurring variable names, shift operators and the MBAs' general structure.

Table 7: QSynth EA dataset results [4], timeout 1 min

Tool	≡	≈	×	Timeout	%
QSynth	N/A	354	146	0	69.0
SiMBA	45	0	455	0	9.0
GAMBA	431	61	8	0	**98.4**

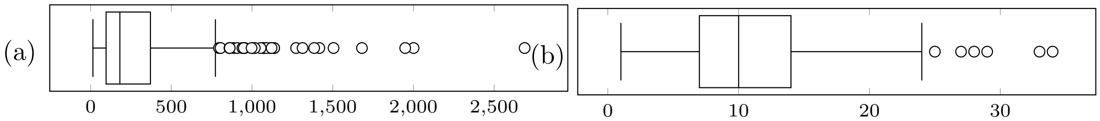
It takes *SiMBA* 0.81 ms on the average (median: 0.59 ms) to solve those expressions that it can solve. *GAMBA* runs 134.18 ms on the average (median: 53.95 ms). Figure 2 demonstrates the high variance of *GAMBA*'s runtimes when applied to the QSynth EA dataset. The maximum runtime is about 3 082.2 ms while most runs take fractions of a second. It is also notable that all equivalences in column ≈ were proved by *GAMBA* itself, i.e., usage of Z3 was not required.

Figure 2: Boxplot of *GAMBA*'s runtimes on QSynth EA



Since this is the most complex dataset, we additionally consider the complexities of the original MBAs as well of the results. Figure 3 shows statistics on the numbers of nodes of the original expressions as well as of their ground truths, indicating that we have a variety of very differently complex expressions.

Figure 3: Boxplots describing the QSynth EA dataset: number of nodes of (a) e and (b) ground truth $e^\star$



In Figure 4 we consider statistics of the results of runs in which *GAMBA* could not derive the exact ground truths as results. Interestingly, the deviance in complexity is in general not larger when *GAMBA* completely fails to show an equivalence. Also notably, *GAMBA*'s results are in rare cases even simpler than the desired ones.

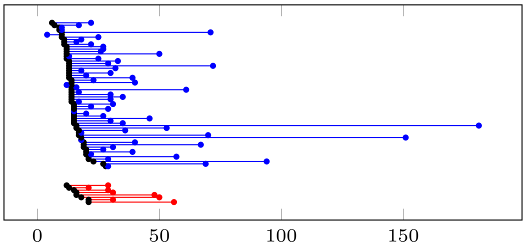
We omit detailed results for the *QSynth Syntia* and *EA-ED* datasets which contain even more complex expressions (up to 5 000 000 characters per expression); the majority of expressions is solvable, with a few timeouts.

6.6 Summary

SiMBA can not only be applied to linear expressions, but also to nonlinear expressions that are reducible to linear ones. We observed that it simplified all expressions in whose ground truths are linear correctly.

Further, we see that *GAMBA* can simplify almost all MBAs to the exact same result as the corresponding ground truth, and it does so quickly (within 50 ms typically, there are few outlines in the datasets). However, *GAMBA* is about ten times slower compared to *SiMBA* on the datasets. The *QSynth EA* dataset is the hardest to simplify. We want to emphasize that even in case of

Figure 4: Deviances in the numbers of nodes from ground truths of equivalent (≈) and failed (×) runs of *GAMBA* on the QSynth EA dataset



failure[7] (×), *GAMBA*'s results are usually drastically reduced compared to the input MBA and quite close to the corresponding ground truths. We have not observed any expression which could not be simplified significantly.

GAMBA's runtimes are, as expected, higher than *SiMBA*'s, but still very practical. Even for linear MBAs, *GAMBA* first has to parse them and figure out that they are linear before they are simplified via a call to *SiMBA*. For nonlinear MBAs, it performs additional refinement steps as well as substitutions and calls *SiMBA* multiple times.

7 Conclusion

In this paper we have extended the algorithm presented in [19] and its range of possible applications. We have seen in Section 6 that it is, in the current state of development, already a powerful tool that can simplify a wide variety of expressions.

Our substitution logic and refinement steps make *GAMBA* solve MBAs which comparable tools cannot simplify. For instance, many tools have problems with large – apparently random – constants, and nonlinear expressions as operands of bitwise operators. Besides, *GAMBA* can solve MBAs whose linear parts are obscured via expansion of products to a certain degree.

We experienced the biggest challenge with MBAs that use a high number of variables and cannot be split into expressions with fewer variables in a meaningful way, and with MBAs that have constants within bitwise operations. The substitution logic may be faced with problems when substituting all constants by variables, and furthermore doing so will obscure the relation between the constants. Hence one may hope that the refinement techniques will make the constants vanish.

We believe *SiMBA* and *GAMBA* are valuable tools for the analysis of code obfuscated with MBA expressions. They are straightforward to use and can be easily deployed in program analysis frameworks.

In spite of the challenges mentioned above, we are convinced to have shown that in general every MBA is solvable, independently of its type or complexity. We summarize our main contributions in this paper:

1. We have extended the linear simplifier *SiMBA* such that it finds linear combinations of an unnegated and a negated bitwise expression as well as of two negated ones, closing the gap to be able to find simplest solutions in all cases for two variables.

2. We have introduced various metrics and exhaustively search the space of all possible solutions in order to find the one minimizing a given metric. We have emphasized that result vectors can often be decomposed in a large number of ways and those may imply solutions of very different complexity.

3. We have extended *SiMBA* to find simple solutions for any number of variables, including more than three.

[7] Note that "failure" means that we cannot verify an MBA's equivalence to the groundtruth using *GAMBA* or Z3. However, we checked the solutions numerically for a large number of inputs.

4. In this course, we have described how to generate bitwise expressions that fit given truth value vectors.

5. We have introduced the powerful, open-source algorithm *GAMBA* for simplifying nonlinear MBAs of any kind, applicable to a wide range of inputs MBAs.

6. In contrast to existing algebraic simplifiers, *GAMBA* is able to simplify expressions that have constants or arithmetic operations within bitwise operations.

7. We have given arguments suggesting that any kind of MBA is simplifiable by algebraic means. *GAMBA* can solve all public datasets known to us and has good performance, as our experiments show.

Data Availability

GAMBA's source code is on Github [20]. Datasets used for experiments are referenced in Subsection 6.

References

[1] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia. Synthesizing the semantics of obfuscated code. In *Proc. 26th USENIX Security Symposium*, pages 643–659, Vancouver, August 2017.

[2] Tim Blazytko and Moritz Schloegel. msynth. `https://github.com/mrphrazer/msynth`, 2020.

[3] Christian Collberg. Tigress. `https://tigress.wtf/`.

[4] Robin David, Luigi Coniglio, and Mariano Ceccato. QSynth. A program synthesis based approach for binary code deobfuscation. In *Workshop on Binary Analysis Research (BAR), NDSS Symposium 2020*, San Diego, February 2020.

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

[6] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based obfuscation. In *Proc. 2nd ACM Workshop on Software PROtection, SPRO'16*, pages 27–38, Vienna, October 2016.

[7] Matteo Favaro and Tim Blazytko. Improving MBA deobfuscation using equality saturation. `https://secret.club/2022/08/08/eqsat-oracle-synthesis.html`, 2022.

[8] Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. NeuReduce. Reducing mixed Boolean-arithmetic expressions by recurrent neural network. In *Findings of the Assoc. for Computational Linguistics*, EMNLP 2020, pages 635–644, Nov. 2020.

[9] Garrett Gu. Hands-free binary deobfuscation with gooMBA. `https://hex-rays.com/blog/deobfuscation-with-goomba`, January 2023.

[10] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, canonicalization and identification of mixed Boolean-arithmetic symbolic expressions. In *GreHack 2016*, Grenoble, France, November 2016.

[11] Binbin Liu, Weijie Feng, Qilong Zheng, Jing Li, and Dongpeng Xu. Software obfuscation with non-linear mixed boolean-arithmetic expressions. In *Proc. Int. Conference on Information and Communications Security, ICICS'21*, volume 12918 of *LNCS*, pages 276–292, Chongqing, China, September 2021. Springer.

[12] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. MBA-Blast. Unveiling and simplifying mixed Boolean-arithmetic obfuscation. In *Proc. 30th USENIX Security Symposium*, pages 1701–1718, Aug. 2021.

[13] Binbin Liu, Qilong Zheng, Jiang Ming, and Dongpeng Xu. An in-place simplification on mixed boolean-arithmetic expressions. *Security and Communication Networks*, 2022:1–14, September 2022.

[14] Edward J. McCluskey. Minimization of Boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[15] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. AI-based blackbox code deobfuscation. Understand, improve and mitigate. *CoRR*, abs/2102.04805, Feb. 2021.

[16] Camille Mougey and Francis Gabriel. DRM obfuscation versus auxiliary attacks. `https://recon.cx/2014/slides/recon2014-21-mougey-camille-francis-gabriel-DRM-obfuscation-versus-auxiliary-attacks-slides.pdf`, June 2014. REcon'14, Montreal.

[17] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, Sep. 2011.

[18] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[19] Benjamin Reichenwallner and Peter Meerwald-Stadler. Efficient deobfuscation of linear mixed boolean-arithmetic expressions. In *Proc. 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man-At-The-End attacks*, CheckMATE'22, pages 19–28, Los Angeles, November 2022.

[20] Benjamin Reichenwallner and Peter Meerwald-Stadler. GAMBA code and dataset. `https://github.com/DenuvoSoftwareSolutions/GAMBA`, 2023.

[21] Microsoft Research. Z3. `https://github.com/Z3Prover/z3`.

[22] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGPLAN Notices*, volume 48, pages 305–316, April 2013.

[23] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. LOKI. Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium*, pages 3055–3073, Boston, Aug. 2022.

[24] Antoine De Schrijver. Automated localisation of a mixed boolean arithmetic obfuscation window in a program binary. Master's thesis, Ghent University, Belgium, 2021.

[25] 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):1–37, March 2017.

[26] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, and Qiaoyan Yu. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In *Proc. 42nd ACM SIGPLAN Int. Conference on Programming Language Design and Implementation, PLDI'21*, pages 651–664, Virtual, Canada, June 2021.

[27] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. Information hiding in software with mixed Boolean-arithmetic transforms. In *Proc. 8th Int. Workshop on Information Security Applications, WISA'07*, volume 4867 of *LNCS*, pages 61–75, Jeju Island, Korea, August 2007. Springer.