Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions [*]

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

Abstract

Mixed Boolean-Arithmetic (MBA) expressions are frequently used for obfuscation. As they combine arithmetic as well as Boolean operations, neither arithmetic laws nor transformation rules for logical formulas can be applied to suitably complex expressions, making MBAs hard to simplify and solve.

In 2019, Liu et al. demystified linear MBAs, leveraging a transformation between the set $B = \{0, 1\}$ of bit values and the set B^n of words of length $n \in \mathbb{N}$ for linear MBAs, originally introduced by Zhou et al. in 2007. With their *MBA-Blast* and *MBA-Solver* algorithms, they outperform existing tools noticably in terms of performance as well as ability to simplify of such MBAs.

We propose a surprisingly simple algorithm called *SiMBA* that improves upon MBA-Blast and MBA-Solver in that it can deobfuscate all linear MBAs, does not miss particularly simple solutions and takes only a fraction of their runtime.

1 Introduction

Mixed Boolean-arithmetic (MBA) transformation is a currently popular technique for code obfuscation introduced in the year 2006 by Zhou et al. [17, 18]. Simple expressions such as constants are replaced by semantically equivalent mixed Boolean-arithmetic expressions in order to make (commonly binary) code harder understandable and thereby help hide secret information such as data and algorithms — e.g., used for watermarking or license checks — via introduction of exaggerated complexity. The goal is to rewrite passages which are easily identifiable in program code as well as in binaries into expressions which are not easily referrable to the original ones.

MBA expressions contain logical as well as arithmetic operations. Due to a bad interaction between those, they cannot be resolved straightforward using any established SAT solvers or mathematical tools which either concentrate on logical or on arithmetic expressions. However, with the rise of methods to generate MBAs, a variety of tools for their deobfuscation, i.e., simplification, or verification are being developed. They use various techniques such as pattern matching (e.g., SSPAM [5]), neural networks (e.g., NeuReduce [7]), bit-blasting (e.g., Arybo [10]), stochastic program synthesis (e.g., Stoke [14], Syntia [1] and Xyntia [12]) or synthesis-based expression simplification (e.g., QSynth [3] and msynth [2]).

Sufficiently complex MBAs are commonly generated by a rewriting technique, iteratively replacing subexpressions by equivalent, more complex ones using a codebook of MBA identities [15]. Obviously, such a codebook can also be used for an attempt to simplify complex MBAs. Since in general MBA expressions cannot be expected to appear in a codebook right away, an SMT solver can be used for equivalence checks against the listed simpler MBAs. MBAs can further be hardened against

[*]Accepted for presentation at the CheckMATE workshop on research of offensive and defensive techniques in the context of Man-At-The-End (MATE) attacks, co-located with ACM Conference on Computer and Communications Security (CCS) 2022, November 7 - 11, 2022.

deobfuscation by applying additional encoding, invertible functions or point functions [18, 15]. This may introduce large constants which pose problems to many deobfuscation tools.

Recently, a first algorithm for fully algebraic attacks has been developed, incorporated in the highly related tools *MBA-Blast* [11] and *MBA-Solver* [16]. These make large progress to the deobfuscation of especially so-called *linear MBAs* as well as of *polynomial MBAs* with small restrictions. As written in their related papers and supported by our own experiments, these tools outperform other existing tools significantly for these classes of MBAs when it comes to simplification success and runtime. They are based on an alternative method for generating MBAs relying on a transformation of linear MBAs between the n-bit space for any $n \in \mathbb{N}$ and the 1-bit space (in other words, between bitwise and logical expressions) first described by Zhou et al. [18] and basically reverse the process for deobfuscation.

Motivated by their approach, we contribute a related, but different algorithm which we call *SiMBA* (for *Simple MBA Simplifier*) and which is characterized by its simplicity, its competitive performance, its genericness derived through an implementation which is independent of the number of variables and the MBAs' complexity as well as the detail that it avoids a full transformation to the 1-bit space, hence does not require a decomposition of an input MBA into terms consisting of bitwise expressions and constant factors. Therefore it allows more flexibility concerning the structure of its input. Its implementation as well as our self-generated datasets for its evaluation are publicly available on Github [13]. Moreover, we give a step-by-step deduction of the fundamental theorems the mentioned peer tools are based on.

In the next section we provide a definition of linear and polynomial MBAs. Additionally we describe how MBAs equivalent to a specific target function, that may itself be a linear MBA, can be generated. On the one hand this is relevant for the generation of MBAs for experiments and on the other hand the theorems presented there also pave the way for our deobfuscation technique that is also used in variations by peer tools. We revisit the groundbreaking theorem of Zhou et al. [18] and provide clear verifications for statements used for the generation of linear MBAs which are aimed at substitute the zero constant, a bitwise expression or even another linear MBA as well as for their deobfuscation.

The most comparable peer tools *MBA-Blast* and *MBA-Solver* which provide efficient deobfuscation algorithms for polynomial MBAs are described in the following section. This section additionally gives an overview of our SiMBA algorithm and its differences to MBA-Blast and MBA-Solver, and provides a proof of its formal foundation. Additionally, we provide results of experiments comparing SiMBA to these tools.

Finally, we point out our main contributions and give an outlook on the simplification of nonlinear MBAs which may use our approach too after identification of its linear parts.

2 Preliminaries

2.1 Linear Mixed Boolean-Arithmetic Expressions

Mixed Boolean-arithmetic expressions mix Boolean expressions with arithmetic ones. As there is obviously a strong connection between logical and bitwise operations, the concept of Boolean expressions is often intermixed with that of bitwise 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 link is crucial for recent findings about MBAs, e.g., a generation algorithm proposed by Zhou et al. [18] and the highly related deobfuscation tools MBA-Blast [11] and MBA-Solver [16].

We prefer the notion of bitwise expressions which fits better to our context. Hence we use the operators & (bitwise conjunction), ^ (bitwise exclusive disjunction), | (bitwise inclusive disjunction) and $\sim$ (bitwise negation) rather than $\wedge$ (logical conjunction), $\oplus$ (logical exclusive disjunction), $\vee$ (logical inclusive disjunction) and $\neg$ (logical negation).

Although we are mainly interested in linear MBAs, we state the — more general — definition of a polynomial MBA first.

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

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

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 $x_1,\dots,x_t$ for $j \in I_j$ and $i \in I$.

A linear MBA is a special kind of a polynomial one where each term consists of only one bitwise expression and an optional constant factor.

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

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

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

These definitions' origin lies in the groundbreaking paper of Zhou et al. [18] in which MBAs are proposed as a promising technique for obfuscation since logical and arithmetic operators do not interact nicely enough for MBAs to be easily resolvable and not much theory on MBAs existed at that time. In that paper, they also propose a much noticed method for generating linear MBAs. It relies on a fundamental relation between Boolean and bitwise expressions.

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

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

be a linear combination of these bitwise expressions with coefficients $a_j \in B^n$ for $j = 1,\dots,s$ and hence a linear MBA. Furthermore let, again for $j = 1,\dots,s$, $\overline{e}_j : B^t \to B$ be the logical expression corresponding to e_j. Enumerate the possible combinations of zeros and ones for the variables by $B_t = \{b_1,\dots,b_{2^t}\}$ 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 $Y = Y_a = (a_1,\dots,a_s)^T$ is a solution of the linear equation system $AY = o$ over B^n, where $o = (0,\dots,0)^T$ is the zero vector in B^{2^t}.

Note that the theorem's original statement only requires that the linear equation system is solvable, but this does not suffice in general. The concrete vector is required to be a solution. As we will see, this theorem is very crucial for the generation of linear MBAs as well as for their deobfuscation, but the latter purpose remained unnoticed for a long time, possibly due to its incorrect formulation.

Apart from that, Zhou et al. [18] also show that each bitwise expression e of an arbitrary number $t \in \mathbb{N}$ of variables has a nontrivial linear MBA representation $e = \sum_{j=1}^{2^t} a_j e_j$ with bitwise expressions $e_j \neq e$ and constants a_j for $j = 1,\dots,2^t$. Moreover, there are infinitely many MBAs in total.

2.2 Generation of Linear Mixed Boolean-Arithmetic Expressions

Theorem 1 implies a method for generating linear MBAs which is very contrary to the codebook approach proposed in other sources. While it suffers a bit from the very specific shape which generated expressions will have, its advantages are that it can choose from an infinite set of MBAs, and a linear MBA always exists for a given input.

For a fixed number t of variables and another fixed number s of bitwise expressions, any matrix $A \in B^{2^t \times s}$ containing only zeros and ones can be used for generating a linear MBA for t variables over B^n for some $n \in \mathbb{N}$ if its columns are linearly dependent.

For given bitwise expressions $e_1, \dots, e_s$ and its truth value matrix A with linearly dependent columns, a solution of the linear equation system $AY = o$ gives the constants a_j for a linear MBA $e = \sum_{j=1}^{s} a_j e_j$ that evaluates to zero for all possible inputs. As a consequence, any of those s bitwise expressions, say, e_j for some $j \in \{1, \dots, s\}$, can be expressed as a linear combination of the other ones by multiplying e by a_j's multiplicative inverse in B^n. Note that if A's columns were linearly independent, the only solution of the equation system would be the zero vector, which is of course not constructive.

Alternatively, one may generate all or a subset of A's columns randomly and construct bitwise expressions fitting the corresponding truth values. This will add more variety, but if all bitwise expressions are generated randomly, it is in general not possible to find an MBA representing a bitwise expression (in place of zero) easily.

This is where the following extension of Theorem 1 comes in handy.

Corollary 1. *Let $n, s, t \in \mathbb{N}$ and let the bitwise expressions $e_1, \dots, e_s$ and e as well as the logical expressions $\overline{e}_1, \dots, \overline{e}_s$ and the matrix A be defined as in Theorem 1. Furthermore, let f be another bitwise expression of t variables and let $\overline{f}$ be its logical equivalent.*

Then $e \equiv f$ if and only if $AY_a = F$, where $Y_a = (a_1, \dots, a_s)^T$ is the vector of e's coefficients and $F = \left(\overline{f}(b_1), \dots, \overline{f}(b_{2^t})\right)^T$ is the vector of f's evaluations on all possible combinations of zeros and ones for the variables, restricted to one bit.

Proof. If $e \equiv f$, then $e_1 := e - f \equiv 0$. Since e_1's truth table matrix is given by $A_1 = (A, F)$, i.e., e's truth table matrix with an appended column containing f's truth values, and its coefficient vector is $Y_1 = (a_1, \dots, a_s, -1)$, it follows from Theorem 1 that $A_1 Y_1 = o$ with $o = (0, \dots, 0)^T$. That is, $\sum_{j=1}^{s} a_j \overline{e}_j(b_i) - \overline{f}(b_i) = 0$ for all $i = 1, \dots, 2^t$. The correctness of this implication follows.

The other one, i.e., that $AY_a = F$ implies that $e \equiv f$, is proven by performing these steps in reversed order. □

This theorem allows one to find a linear MBA representing an arbitrary bitwise expression f via a method equivalent to that described above.

Finally, we generalize this even more for affine integer combinations of bitwise expressions.

Corollary 2. *Let $n, s, t \in \mathbb{N}$ and let the bitwise expressions $e_1, \dots, e_s$ and e as well as the logical expressions $\overline{e}_1, \dots, \overline{e}_s$ and the matrix A be defined as in Theorem 1. Furthermore, let, for some $m \in \mathbb{N}$, $f = \sum_{k=1}^{m} \alpha_k f_k + \beta$ be an affine combination of bitwise expressions of t variables with $\alpha_1, \dots, \alpha_m, \beta \in B^n$. Additionally, let $g = \sum_{k=1}^{m} \alpha_k \overline{f}_k - \beta$, where we write $-\beta$ for an integer that sums up to zero with β in the modular field B^n and $\overline{f}_k$ is the logical equivalent to f_k for $k = 1, \dots, m$.*

Then $e \equiv f$ if and only if $AY_a = F$, where $Y_a = (a_1, \dots, a_s)^T$ is the vector of e's coefficients and $F = (g(b_1), \dots, g(b_{2^t}))^T$ is the vector of g's evaluations on all possible combinations of zeros and ones for the variables.

Proof. The proof works similarly as that of Corollary 1. In order to apply Theorem 1, we want to express the constant term as a multiple of a bitwise expression which evaluates to the n-bit number which has ones everywhere, i.e., $\sum_{i=1}^{n} 2^{i-1} \times 1 = 2^n - 1 \equiv -1 \mod 2^n$. That is, the coefficient β's sign is reversed in the application of the theorem. The rest follows. $\square$

Corollary 2 allows us to find arbitrarily complex MBAs with a nearly arbitrary number of variables representing a large variety of affine combinations of bitwise expressions. We list some simple examples with two variables here:

$$\begin{aligned}
&x + y \to 2\left((x \mathbin{\&} y) \mid ({\sim}x \mathbin{\&} {\sim}y)\right) - 2\left({\sim}x \mathbin{\&} y\right) \\
&\qquad + 3\left(({\sim}x \mathbin{\&} y) \mid (x \mathbin{\&} {\sim}y)\right) - 2 \cdot {\sim}y \\
&3\,735\,936\,685 \cdot {\sim}x \to -3\,735\,936\,685\,(x \mathbin{\&} {\sim}y) + 3\,735\,936\,685\,(y \mid {\sim}x) \\
&\qquad + 3\,735\,936\,685\left(({\sim}x \mathbin{\&} y)|(x \mathbin{\&} {\sim}y)\right) - 3\,735\,936\,685\,y \\
&3\,735\,936\,685\,(x \mathbin{^\wedge} y) + 49\,374 \to 3\,735\,911\,998 \cdot {\sim}x - 24\,687\,(x \mid {\sim}y) \\
&\qquad - 3\,735\,936\,685\,(y \mid {\sim}x) + 3\,735\,911\,998\,(y \mid x)
\end{aligned}$$

What the corollary additionally suggests is that an affine combination of bitwise expressions is a linear MBA; that is, there is no need for a definition of, say, affine MBAs. Against this background, the corollary is related to Theorem 1 in [16] while differently and probably more constructively proven. It states that two linear MBAs are equivalent if and only if their *signature vectors* coincide. We restate it for the sake of completeness:

Corollary 3. *Let e_1 and e_2 be linear MBAs over words of the same length n with truth table matrices A_1 and A_2, resp., and coefficient vectors Y_1 and Y_2, resp. Then $e_1 \equiv e_2$ if and only if $A_1Y_1 = A_2Y_2$ or, equivalently, their linear combinations of logical expressions corresponding to their bitwise expressions evaluate to the same values for all b_i, $i = 1, \dots, 2^t$ as defined previously.*

3 Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions

The mixed Boolean-arithmetic transform technique is commonly considered to be a powerful obfuscation method due to the incompatibility of its operands. However, Theorem 1 and specifically Corollary 3 prepare an algebraic method which may circumvent this issue. This method has been first elaborated and implemented in the tools *MBA-Blast* and *MBA-Solver* which outperform existing tools significantly regarding success frequency and runtime at the deobfuscation of MBAs that have a specific shape.

3.1 MBA-Blast and MBA-Solver

MBA-Blast and MBA-Solver are two highly related tools for efficient deobfuscation of MBAs. Both are available in a prototype state on Github [8, 9]. While the former can simplify polynomial MBAs, the latter applies a more efficient approach for those and attempts to resolve nonpolynomial MBAs as well. However, for that it needs a hint of a polynomial MBA included in a nonpolynomial MBA, whose replacement by a variable would cause it to be polynomial, as well as an additional input, making it irrelevant for practical applications at least in this prototype state.

The tools as available on Github strongly suffer from the requirement for prior knowledge about MBAs to resolve. E.g., MBAs have to meet exactly the canonical representation used in Definition 1 and the user has to know whether they are linear, polynomial or nonpolynomial (with additional required knowledge in the latter case). A check for MBAs to be linear or polynomial, resp., is not performed.

The tools are implemented for simplifying MBAs working on two to four variables. First of all, each space of bitwise expressions using $t \in \mathbb{N}$ variables has a basis of 2^t expressions (e.g., 16 expressions for $t = 4$ variables) each bitwise expression can be represented as a linear combination of. The MBA-Blast algorithm starts by replacing each bitwise expression occuring in an input MBA by such a linear combination. This is done via their evaluation for all possible combinations of truth values for the variables and hence leveraging Theorem 1. Although this crucial finding dates back to the year 2007 [18], it has seemingly never been recognized to be readily usable for deobfuscation before due to its incorrect statement — see [4] or also the MBA-Blast paper [11] whose authors claim to be the first to prove it.

In contrast to that, MBA-Solver computes a so-called *signature vector* for a whole linear MBA leveraging Corollary 3, again via an evaluation of its bitwise expressions for all possible combinations of truth values, but here those are immediately multiplied by the bitwise expressions' coefficients and summed up while the linear combinations have to be summed up with MBA-Blast in order to derive a linear combination for a whole MBA. Hence, MBA-Solver saves this step, and consequently significant runtime.

More involved simplification steps, including simplification using Python's `SymPy` package, are necessary for nonlinear MBAs. To the best of our understanding, different bases are used for MBA-Blast and MBA-Solver. While the former uses, e.g., for two variables the vector $(x, y, x \,\&\, y, -1)$ as a basis, the latter uses $({\sim}(x \mid y), {\sim}(x \mid {\sim}y), x \,\&\, {\sim}y, x \,\&\, y)$. The reason for this discrepancy is not known to the authors, but we assume that this is just an exemplary base and a full implementation of this prototype would, as indicated in [16], try various bases and choose the simplest result.

Alternatively, if the resulting signature vector, now also computed for a whole linear MBA with MBA-Blast, is nice enough and corresponds to one single bitwise expression, this one is the simplification result. In this case this expression is found in a lookup table listing standard representations of bitwise expressions for all possible combinations of the variables' truth values.

While the step of finding a linear combination of basis expressions can be implemented genericly to work for an arbitrary number of variables, usage of a lookup table for finding single bitwise expressions is only possible for a low number of variables. Note that a lookup table for $t = 4$ variables has $2^{2^4} = 65\,536$ entries, and one for $t = 5$ variables would have $2^{2^5} = 4\,294\,967\,296$ entries.

Due to this restriction and since it cannot be expected in general that an MBA can be resolved to one single bitwise expression, the tools might miss the nicest solutions in many cases. However, this may be acceptable since their main purpose is to make expressions easier for verification using SMT solvers [9] and linear combinations might be nice enough.

For input expressions which satisfy the requirements on their structure, MBA-Blast and MBA-Solver indeed clearly outperform other existing tools regarding success rate and runtime.

3.2 Our Algorithm

3.2.1 Overview

As MBA-Blast and MBA-Solver, SiMBA is written in `Python`. Although it was developed without knowledge of MBA-Solver's exact technique, it uses a rather similar technique also involving a computation of what one may call a *signature vector* in the first step. However, while MBA-Blast and MBA-Solver use Corollaries 1 and 3, resp., and hence operate in the 1-bit space $B = \{0, 1\}$, we do not fully perform this transformation from B^n (for $n \in \mathbb{N}$) to B. Rather we immediately evaluate the whole linear MBA for the combinations of truth values and hence get vectors which live in B^n. The following theorem provides the formal basis for that.

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 $e(b_i) = f(b_i)$ for all $b_i \in B_t$, $i = 1, \ldots, 2^t$, as defined previously.*

Note that the difference to Corollary 3 is that we evaluate the bitwise expressions rather than their logical (1-bit) equivalents for the b_i's and hence get n-bit results. That implies that we do not have to decompose a linear MBA into its terms and factors, but we can evaluate it as a whole.

Proof. We only have to prove one direction since the other one is trivial. We assume that $e(b_i) = f(b_i)$ for all b_i, $i = 1, \dots, 2^t$ and write $e = \sum_{j=1}^{s} a_j e_j$ and $f = \sum_{j=1}^{m} \alpha_j f_j$ for $s, m \in \mathbb{N}$, $a_1, \dots, a_s$, $\alpha_1, \dots, \alpha_m \in B^n$ and bitwise expressions $e_1, \dots, e_s$, $f_1, \dots, f_m$.

We have for all $i \in \{1, \dots, 2^t\}$ that $\sum_{j=1}^{s} a_j e_j(b_i) = \sum_{j=1}^{m} \alpha_j f_j(b_i)$. Now let $\overline{e}_j$ be the logical equivalent of e_j for $j = 1, \dots, s$ and $\overline{f}_j$ be the logical equivalent of f_j for $j = 1, \dots, m$. Furthermore, for $i = 1, \dots, 2^t$, we use the same notation for b_i with one bit as for b_i with n bits, but we note that the n-bit version has zeros in the additional $n-1$ bits of its entries.

We want to show that $\sum_{j=1}^{s} a_j \overline{e}_j(b_i) = \sum_{j=1}^{m} \alpha_j \overline{f}_j(b_i)$ for all $i \in \{1, \dots, 2^t\}$. Then we may use Corollary 3. We have the following for any $i \in \{1, \dots, 2^t\}$ after already interchanging the sums:

$$\begin{aligned}\sum_{j=1}^{s} a_j \overline{e}_j(b_i) + \sum_{k=2}^{n} 2^{k-1} \sum_{j=1}^{s} a_j \overline{e}_j(o)& \\ = \sum_{j=1}^{m} \alpha_j \overline{f}_j(b_i) + \sum_{k=2}^{n} 2^{k-1} \sum_{j=1}^{m} \alpha_j \overline{f}_j(o),\end{aligned}$$

where we write o for the zero vector in B^t and the sum $\sum_{k=2}^{n} 2^{k-1}$ is obviously constant ($2^n - 2$, in more detail).

If it holds that $\sum_{j=1}^{s} a_j \overline{e}_j(o) = \sum_{j=1}^{m} \alpha_j \overline{f}_j(o)$, the proof is finished. We consider the equation for the specific $b_{i'} \in B_t$ which assigns zero to every variable:

$$(2^n - 1) \sum_{j=1}^{s} a_j \overline{e}_j(o) = (2^n - 1) \sum_{j=1}^{m} \alpha_j \overline{f}_j(o).$$

This shows that $\sum_{j=1}^{s} a_j \overline{e}_j(o) = \sum_{j=1}^{m} \alpha_j \overline{f}_j(o)$. □

Depending on the number of variables, the SiMBA algorithm is comprised of two or three steps described subsequently in more detail:

1. For an input MBA e, a signature vector F is derived by evaluating e for all possible combinations of zeros and ones for the variables.
2. Based on F, the input MBA is transformed into a linear combination of base bitwise expressions, using Theorem 2. This is in any case a valid, but not necessarily optimal solution.
3. If the resulting linear combination uses fewer than three variables, various attempts to find a simpler expression via lookup tables are performed.

The main advantage of our approach is that due to the immediate evaluation, one does not have to decompose an MBA into terms of bitwise expressions and their coefficients and hence does not require a concrete structure of input expressions. Optionally a check for the MBA's linearity can be performed. Moreover we expect a better performance especially for complex MBAs with a higher number of variables.

3.2.2 Computing the Signature Vector

In order to be able to use Theorem 2, we evaluate an input expression e on all possible combinations of zeros and ones for the variables and obtain a vector $F = e(b_1, \dots, b_{2^t})$. In order to be most generic, for $k \in \{1, \dots, 2^t\}$ and $i \in \{1, \dots, t\}$, b_k assigns the value 1 to the variable x_i if k's remainder after a division by 2^i is larger than 2^{i-1}:

$$\begin{aligned} b_1 &= (0,0,0,\dots,0), \\ b_2 &= (1,0,0,\dots,0), \\ b_3 &= (0,1,0,\dots,0), \\ b_4 &= (1,1,0,\dots,0), \\ &\vdots \\ b_{2^t} &= (1,1,1,\dots,1). \end{aligned}$$

As an example, we consider the simple linear MBA $e(x,y) = 3\,735\,936\,685\,(x \mathbin{^\wedge} y) + 49\,374$. In B, we would get a vector

$$(e'(0,0), e'(1,0), e'(0,1), e'(1,1)) = (0,1,1,0)$$

for the bitwise part $e'(x,y) = x \mathbin{^\wedge} y$. This vector would have to be multiplied by the coefficient and the constant would have to be subtracted (see Corollary 2) in order to obtain a vector F_1'. In our setting, we immediately compute

$$\begin{aligned} F_1 &= (e(0,0), e(1,0), e(0,1), e(1,1)) \\ &= (49\,374, 3\,735\,986\,059, 3\,735\,986\,059, 49\,374) \\ &= F_1' + 2 \times 49\,374. \end{aligned}$$

We see another difference when we consider the negation $f(x) = {\sim}x$. Rather than using the signature vector

$$F_2' = (f(0), f(1)) = (1,0),$$

we use

$$F_2 = (f(0), f(1)) = (-1,-2) = (-0-1, -1-1),$$

where we evaluate f on B in the former evaluation and on B^n in the latter one.

3.2.3 Finding a Linear Combination

The next step is the derivation of a linear combination of basis elements for bitwise expressions with $t \in \mathbb{N}$ variables. While in theory any basis can be used, we want to describe a very generic approach. If we enumerate the variables as $x_1, \dots, x_t$, the following is a basis for t variables:

$$\begin{aligned} (1,& \\ &x_1, \dots, x_t, \\ &x_1 \mathbin{\&} x_2, \dots, x_1 \mathbin{\&} x_t, x_2 \mathbin{\&} x_3, \dots, x_2 \mathbin{\&} x_t, \dots, x_{t-1} \mathbin{\&} x_t, \\ &x_1 \mathbin{\&} x_2 \mathbin{\&} x_3, x_1 \mathbin{\&} x_2 \mathbin{\&} x_4, \dots, x_{t-2} \mathbin{\&} x_{t-1} \mathbin{\&} x_t, \\ &\vdots \\ &x_1 \mathbin{\&} \cdots \mathbin{\&} x_t). \end{aligned}$$

It has the following truth table A:

$$\begin{array}{c} \\ \scriptstyle(0,0,0,\ldots,0) \\ \scriptstyle(1,0,0,\ldots,0) \\ \scriptstyle(0,1,0,\ldots,0) \\ \scriptstyle(1,1,0,\ldots,0) \\ \scriptstyle(0,0,1,\ldots,0) \\ \scriptstyle(1,0,1,\ldots,0) \\ \\ \scriptstyle(1,1,1,\ldots,1) \end{array} \begin{array}{c} \begin{array}{cccccccc} \scriptstyle 1 & \scriptstyle x_1 & \scriptstyle x_2 & \scriptstyle\cdots & \scriptstyle x_1\&x_2 & \scriptstyle x_1\&x_3 & \scriptstyle\cdots & \scriptstyle x_1\&\cdots\&x_t \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 1 & 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 1 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & 1 & 1 & \cdots & 1 \end{pmatrix} \end{array}$$

If we now want to solve a linear equation system $AY = F$, where F is a vector derived in the first step, we do not need much linear algebra and no utilities. If we eliminate A's columns and rows wisely, we permanently have a row which only has one 1 in it until we have computed all coefficients in Y.

In more detail, the first row in which we have a 1 for a variable x_i for $i \in \{1, \ldots, t\}$ is the $(2^{i-1} + 1)$-th one and the first row in which we have a 1 for a conjunction $x_{i_1} \,\&\, \cdots \,\&\, x_{i_m}$ with $i_1 < \cdots < i_m$ for $i_j \in \{1, \ldots, t\}$, $j \in \{1, \ldots, m\}$, $m \in \mathbb{N}$ is the row number $\sum_{j=1}^{m} 2^{i_j - 1} + 1$. These numbers indicate F's entries which we find conjunctions' coefficients in. Similarly, we constantly have to apply the corresponding manipulations for all of F's entries with indices of rows in which A's column has a 1 too.

This approach cannot fail if we handle the conjunctions in an order depending on their numbers of variables, starting with the constant in row one. As soon as this is finished, we have a first viable solution in terms of a linear combination of base expressions.

Returning to the first example, F_1's first entry 49 374 is the constant term which has to be subtracted from F_1's other entries to get $(0, 3\,735\,936\,685, 3\,735\,936\,685, 0)$. Now, the coefficient for both variables — we call them x and y again — is 3 735 936 685, and the vector is modified to $(0, 0, 0, -2 \times 3\,735\,936\,685)$. This implies the following linear combination:

$$\begin{aligned} e(x, y) \equiv\ & 49\,374 + 3\,735\,936\,685\, x + 3\,735\,936\,685\, y \\ & - 7\,471\,873\,370\, (x \,\&\, y). \end{aligned}$$

3.2.4 Finding a Simpler Solution

In order not to miss very simple results, we perform an alternative refinement attempt similar to that of MBA-Blast and MBA-Solver, but more involved. For that we use truth tables for $1 \le t \le 3$. If an input expression has more than three variables, but not all of them appear in the linear combination, we can reduce the number of variables and potentially perform this refinement step too.

As indicated before, we cannot identify a negation just by flipped bits, but we can use the identity $\sim\!x = -x - 1$. We perform the following attempts to find a result which has fewer terms than the linear combination in the following order using a lookup table and finish as soon as an attempt is successful:

1. If all entries of F coincide, we have a constant expression.

   Example for $t = 2$: The vector

   $$(49\,374, 49\,374, 49\,374, 49\,374)$$

   obviously corresponds to

   $$e(x, y) = 49\,374.$$

2. If F has two unique entries and its first entry is zero, we replace the nonzero element a by 1, find the lookup table's entry for the corresponding truth vector and multiply the found expression by a.

   Example for $t = 2$: The vector

   $$(0, 49\,374, 49\,374, 0)$$

corresponds to

$$e(x, y) = 49\,374\,(x \mathbin{\char`\^} y).$$

3. If F has two unique entries a and b, both of them are nonzero, w.l.o.g., $b \equiv 2a \mod 2^n$, and F's first entry is a, we can express the result in terms of a negated single expression. We replace all occurences of a by zeros and that of b by ones, find the corresponding expression in the lookup table, negate it, and multiply it by $-a$.

   Example for $t = 2$: For the vector

   $$(49\,374, 98\,748, 98\,748, 49\,374),$$

   we have $a = 49\,374$ and $b = 98\,748$. The vector $(0, 1, 1, 0)$ corresponds to $x \mathbin{\char`\^} y$. Hence we have a result

   $$e(x, y) = -49\,374 \cdot {\sim}(x \mathbin{\char`\^} y).$$

4. If F has two unique entries a and b, but the previous cases do not apply, and F's very first entry is a, we first identify a as the constant term. Then we find an expression with ones exactly where F has the entry b in the lookup table, multiply it by $b - a$ and add the term to the constant.

   Example for $t = 2$: For the vector

   $$(49\,374, 3\,735\,936\,685, 3\,735\,936\,685, 49\,374),$$

   we have a constant $a = 49\,374$ and, with $b = 3\,735\,936\,685$, relate the vector $(0, b, b, 0)$ to $x \mathbin{\char`\^} y$ and arrive at the expression

   $$e(x, y) = 49\,374 + 3\,735\,887\,311\,(x \mathbin{\char`\^} y).$$

5. If F has two unique nonzero entries a and b and its first one is zero, we split it into two vectors with ones where F has entries a or b, resp., find the corresponding expressions in the lookup table, multiply them by a and b, resp., and add the terms together.

   Example for $t = 2$: We split the vector

   $$(0, 3\,735\,936\,685, 3\,735\,936\,685, 49\,374)$$

   into the vectors $(0, 0, 0, a)$ and $(0, b, b, 0)$ with $a = 49\,374$ and $b = 3\,735\,936\,685$, relate the former to $x \,\&\, y$ and the latter to $x \mathbin{\char`\^} y$ and arrive at the expression

   $$e(x, y) = 49\,374\,(x \,\&\, y) + 3\,735\,936\,685\,(x \mathbin{\char`\^} y).$$

6. If F has three unique nonzero entries a, b and c and its first one is 0, we try to express one of them as a sum of the others modulo 2^n, e.g., $a \equiv b + c$. In that case we split F into two vectors with ones where F has entries b or c, resp., or a, find the corresponding expressions in the lookup table, multiply them by b and c, resp., and add the terms together.

   Example for $t = 2$: We split the vector

   $$(0, 3\,735\,936\,685, 3\,735\,887\,311, 49\,374)$$

   into the vectors $(0, a, a, 0)$ and $(0, b, 0, b)$ with $a = 3\,735\,887\,311$ and $b = 49\,374$, relate the former to $x \mathbin{\char`\^} y$ and the latter to x and arrive at the expression

   $$e(x, y) = 3\,735\,887\,311\,(x \mathbin{\char`\^} y) + 49\,374\,x.$$

7. If F has three unique nonzero entries a, b and c, its first one is 0 and the previous case does not apply, we split it into three vectors with ones where F has entries a, b or c, resp., find the corresponding expressions in the lookup table, multiply them by a, b and c, resp., and add the terms together.

   Example for $t = 2$: We split the vector

   $$(0, 49\,374, 3\,735\,936\,685, 201)$$

   into the vectors $(0, a, 0, 0)$, $(0, 0, b, 0)$ and $(0, 0, 0, c)$ with $a = 49\,374$, $b = 3\,735\,936\,685$ and $c = 201$, relate them to $x \,\&\, {\sim}y$, ${\sim}(x \mid {\sim}y)$ and $x \,\&\, y$ and arrive at the expression

   $$\begin{aligned} e(x, y) = {} & 49\,374\,(x \,\&\, {\sim}y) + 3\,735\,936\,685 \cdot {\sim}(x \mid {\sim}y) \\ & + 201\,(x \,\&\, y), \end{aligned}$$

   but we neglect it since it is not simpler than the linear combination

   $$49\,374\,x + 3\,735\,936\,685\,y + 201\,(x \,\&\, y).$$

8. If F has four unique nonzero values and its first one is nonzero, proceed as above after subtracting the first entry, i.e., the constant term, and add that to the final result.

   Note for $t = 2$: That is not relevant for $t = 2$ since the resulting term would never be simpler than the linear combination of base expressions, but it may be for $t = 3$.

This list of attempts can in theory be extended, but a linear combination of a higher number of bitwise terms from the lookup table might soon appear to be more complex than a linear combination of an even higher number of very simple base expressions. For $t = 2$, the algorithm will definitely find a simplest result.

3.3 Verification and Comparison

In order to classify our algorithm *SiMBA*, we perform four experiments. In the first, we use a dataset of 10 000 linear MBAs with a varying number of variables provided by the Github repository of *NeuReduce* [6] to verify correctness and universality.

Thereafter, we compare the algorithm with the most relevant peer tools. We can confirm the results of [11] and [16] that Arybo, SSPAM and Syntia do not reliably simplify MBAs in general and, if they do, need significantly more time. Hence, we are content with a comparison to MBA-Blast and MBA-Solver only. Corresponding to those, we expect that they do similar things, but MBA-Solver is more efficient since it saves a simplification step. Especially for expressions which can be simplified to one single bitwise expression, the first step is omitted completely.

In the second experiment, we use MBA-Blast (partially), MBA-Solver and SiMBA to simplify all linear MBAs provided by the former's Github repository [9]. In the third one, we run experiments on various MBAs that we generated for simple affine combinations of bitwise expressions using Corollary 2. The final experiment indicates that an affine output encoding does not effect SiMBA's success rate and runtime.

All experiments are performed on a Debian Bullseye virtual machine run on an Intel Core i7-12700K CPU and 3.6 GHz. The runtime was measured using `Python 3.9` with the `time` package. Furthermore, we use $n = 64$ bits in all experiments.

3.3.1 Verification on NeuReduce's Dataset

In a first experiment, we run SiMBA on NeuReduce's test dataset containing 10 000 expressions with two, three, four or five variables.

We see in Table 1 that the entire dataset could be solved by our algorithm, meaning that each expression was simplified to the corresponding simpler expression also contained in the dataset. Moreover, the results show that the runtime only increases moderately with the number of variables and even MBAs with five variables can be simplified fast.

	Total	**Solved**	**Runtime**
2 variables	4 000	4 000	0.00024 s
3 variables	4 560	4 560	0.00069 s
4 variables	441	441	0.00084 s
5 variables	999	999	0.00147 s

Table 1: Verification of SiMBA on the dataset of linear MBAs provided by NeuReduce's Github repository [6]

3.3.2 Comparison on MBA-Solver's Dataset

In Table 2, we compare SiMBA to MBA-Blast and MBA-Solver on the dataset provided by the latter's Github repository. Here we compare the results of the simplification by either tool with the corresponding simpler expression that is also provided in this dataset. Since all tools simplify the complex MBAs and their corresponding simpler version to exactly the same expressions in all cases, we do not have to verify their equivalence using any SMT solver. To some degree we have to rely on the MBAs' correct generation by the dataset providers because some MBAs are too complex for a verification using any SMT solver without prior simplification. The dataset includes 1 008 expressions with two, three or four variables.

Tool	**Average runtime**		
	2 variables (551 expr.)	**3 variables** (350 expr.)	**4 variables** (107 expr.)
MBA-Blast	0.02501 s	0.06726 s	—
MBA-Solver	0.00047 s	0.00121 s	0.14362 s
SiMBA	0.00024 s	0.00116 s	0.00257 s

Table 2: Comparison of MBA-Blast, MBA-Solver and SiMBA on the dataset of linear MBAs provided by MBA-Solver's Github repository [9]

We see that our algorithm runs significantly faster than MBA-Blast and also faster — especially for four variables — than MBA-Solver on the average. Unfortunately we cannot run MBA-Blast for four variables since it is lacking any implementation for that case.

3.3.3 Comparison on Self-Generated MBAs

Leveraging Corollary 2, we have generated 1 000 different MBAs for each of the following very simple expressions and a varying number of variables:

$$\begin{aligned}
e_1(x,y) &= x + y,\\
e_2 &= 49\,374,\\
e_3(x) &= 3\,735\,936\,685\,x + 49\,374,\\
e_4(x,y) &= 3\,735\,936\,685\,(x \mathbin{^\wedge} y) + 49\,374,\\
e_5(x) &= 3\,735\,936\,685 \cdot {\sim}x.
\end{aligned}$$

We then simplified those expressions with the various tools. Previously we had to modify MBA-Blast as well as MBA-Solver in order to accept our input expressions without impacting their logic or runtime. Specifically we had to adapt the structure of our input expressions in order to use the canonical representation of linear MBAs and we had to adapt the variable names.

Most importantly, we had to make sure that occuring constants are reduced modulo 2^n, where n is the number of bits. MBA-Blast and MBA-Solver do not care about this number of bits, but this has two serious drawbacks: On the one hand we might not recognize the equality of expressions which are equivalent for a certain number of bits, but not without any modular reduction. On the other hand, the numeric `NumPy` types internally use `C` integers and throw an error if an overflow occurs for them.

Expr.	**Average runtime of MBA-Blast**	
	2 variables	**3 variables**
e_1	$0.02695\,s$	$0.05366\,s$
e_2	$0.02291\,s$	$0.04833\,s$
e_3	$0.02851\,s$	$0.05753\,s$
e_4	$0.03602\,s$	$0.05699\,s$
e_5	$0.02782\,s$	$0.05459\,s$

Table 3: Runtime of MBA-Blast on 1 000 MBAs generated for five expressions

MBA-Blast simplifies, independently of the number of variables, all expressions for e_1, e_2, e_3 and e_5 — partially due to our modifications — exactly to those simpler expressions while expressions for $e_4(x,y) = 3\,735\,936\,685\,(x \mathbin{\char`\^} y) + 49\,374$ are simplified to

$$\begin{aligned} 3\,735\,936\,685\,x + 3\,735\,936\,685\,y + 18\,446\,744\,066\,237\,678\,246\,(x \mathbin{\&} y) \\ + 49\,374 \end{aligned}$$

since MBA-Blast only finds simple terms if an expression consists of only one term and this is the linear combination of base expressions. Table 3 shows that MBA-Blast has a satisfying and stable runtime for two and three variables while it can, in the available implementation, not be used for a higher number of variables.

Expr.	**Average runtime of MBA-Solver**		
	2 variables	**3 variables**	**4 variables**
e_1	$0.00074\,s$	$0.00091\,s$	$0.12761\,s$
e_2	$0.00052\,s$	$0.00092\,s$	$0.12352\,s$
e_3	$0.00026\,s$	$0.00150\,s$	$0.12802\,s$
e_4	$0.00041\,s$	$0.00160\,s$	$0.12683\,s$
e_5	$0.00072\,s$	$0.00134\,s$	$0.12871\,s$

Table 4: Runtime of MBA-Solver on 1 000 MBAs generated for five expressions

MBA-Solver simplifies, independently of the number of variables, all expressions for e_2 and e_5 — again partially due to our modifications — exactly to those simpler expressions. Unfortunately, due to their basis choice, the simple expression $e(x,y) = x+y$ is missed, and "simplified", depending on the number of variables, to one of the following expressions:

$$\begin{aligned}
&1 \cdot {\sim}(x \mid {\sim}y) + 1\,(x \mathbin{\&} {\sim}y) + 2\,(x \mathbin{\&} y) \\
&1 \cdot {\sim}(x \mid ({\sim}y \mid z)) + 1 \cdot {\sim}({\sim}x \mid (y \mid z)) + 2 \cdot {\sim}({\sim}x \mid ({\sim}y \mid z)) \\
&\qquad + 1\,({\sim}x \mathbin{\&} (y \mathbin{\&} z)) + 1\,(x \mathbin{\&} ({\sim}y \mathbin{\&} z)) + 2\,(x \mathbin{\&} (y \mathbin{\&} z)) \\
&1 \cdot {\sim}(x \mid ({\sim}y \mid (z \mid t))) + 1 \cdot {\sim}({\sim}x \mid (y \mid (z \mid t))) \\
&\qquad + 2 \cdot {\sim}({\sim}x \mid ({\sim}y \mid (z \mid t))) + 1 \cdot {\sim}(x \mid ({\sim}y \mid ({\sim}z \mid t))) \\
&\qquad + 1 \cdot {\sim}({\sim}x \mid (y \mid ({\sim}z \mid t))) + 2 \cdot {\sim}({\sim}x \mid ({\sim}y \mid ({\sim}z \mid t))) \\
&\qquad + 1\,({\sim}x \mathbin{\&} (y \mathbin{\&} ({\sim}z \mathbin{\&} t))) + 1\,(x \mathbin{\&} ({\sim}y \mathbin{\&} ({\sim}z \mathbin{\&} t))) \\
&\qquad + 2\,(x \mathbin{\&} (y \mathbin{\&} ({\sim}z \mathbin{\&} t))) + 1\,({\sim}x \mathbin{\&} (y \mathbin{\&} (z \mathbin{\&} t))) \\
&\qquad + 1\,(x \mathbin{\&} ({\sim}y \mathbin{\&} (z \mathbin{\&} t))) + 2\,(x \mathbin{\&} (y \mathbin{\&} (z \mathbin{\&} t)))
\end{aligned}$$

Using Z3, it takes 0.06561 seconds to verify the equivalence to $e_1(x,y) = x+y$ for the first result, 1.01100 seconds for the second one and 16.63006 seconds for the third one.

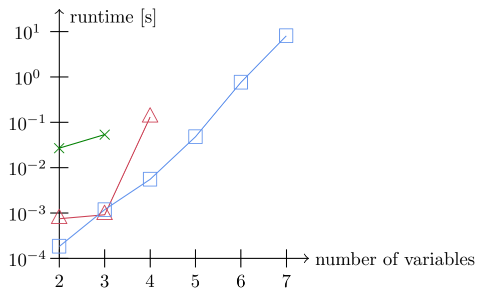

Figure 1: Comparison of the runtimes of MBA-Blast ($\times$), MBA-Solver ($\triangle$) and SiMBA ($\square$) on a logarithmic scale for MBAs for e_1

Even more complex results are obtained for e_3 and for e_4. It is of course not satisfying to get such complex expressions for very simple ones. However, as already mentioned, if MBA-Solver would try different bases as well — and especially $(x, y, x \,\&\, y, -1)$ — it would find a simpler solution. Apart from that, as can be seen in Table 4, MBA-Solver works particularly fast for three variables and much slower for four variables. A possible reason is that the tool has to parse a very large lookup table.[1] Unfortunately we cannot run the tool for higher numbers of variables since its public implementation is not generic enough and restricted to up to four variables.[2]

Expr.	**Average runtime of SiMBA**		
	2 variables	**3 variables**	**4 variables**
e_1	0.00019 s	0.00117 s	0.00550 s
e_2	0.00010 s	0.00045 s	0.00528 s
e_3	0.00020 s	0.00109 s	0.00544 s
e_4	0.00024 s	0.00100 s	0.00564 s
e_5	0.00022 s	0.00108 s	0.00543 s

Table 5: Runtime of SiMBA on 1 000 MBAs generated for five expressions

For SiMBA, all simplified expressions were exactly equal to the corresponding simple expressions. As Table 5 suggests, it has a very competitive runtime and outperforms especially MBA-Blast. The algorithms' runtimes for e_1 are additionally compared in Figure 1; they would be similar for other expressions. Unfortunately we are missing a comparison to MBA-Solver for more than four variables as well as an explanation for MBA-Solver's bad performance for four variables.

3.3.4 Encoded MBAs

In order to make MBAs in general harder to solve, its inputs and/or its output can be encoded using specific functions. We provide some experiments on MBAs whose outputs are encoded using affine functions $f : B^n \to B^n$, $f(x) = ax + b$ where $a, b \in B^n$ are randomly determined. Note that MBA-Blast and MBA-Solver cannot deal with this encoding since everything would have to be multiplied out in order to have the canonical representation as stated in Definition 2, and that input encoding would cause e_4 and e_5 to be nonpolynomial.

In Table 6, we consider runtimes of the simplification of encoded versions of the expressions used in the previous section, i.e.,

$$e_1'(x, y) = a(x + y) + b \text{ for random } a, b \in B^{64}$$

[1]Further experiments suggest that outsourcing the parsing to a preprocessing step would save about 65 per cent of the runtime for four variables. Furthermore, omitting the search for a solution using exactly one bitwise expression would make (nearly) the whole lookup table obsolete and save more than 98 per cent of the runtime in total.

[2]For five variables, we generated a lookup table only containing the $2^5 = 32$ base expressions, disabled the search for a solution using exactly one bitwise expression, which would require the whole lookup table, and performed further adaptions to verify that MBA-Solver has a potential to run fast also for $t = 5$.

and equivalently for $e'_2, \dots, e'_5$.

Expr.	**Average runtime of SiMBA**		
	2 variables	**3 variables**	**4 variables**
e'_1	$0.00020\,s$	$0.00118\,s$	$0.00484\,s$
e'_2	$0.00010\,s$	$0.00029\,s$	$0.00481\,s$
e'_3	$0.00021\,s$	$0.00125\,s$	$0.00460\,s$
e'_4	$0.00020\,s$	$0.00098\,s$	$0.00516\,s$
e'_5	$0.00015\,s$	$0.00110\,s$	$0.00547\,s$

Table 6: Runtime of SiMBA on 1 000 MBAs with output encoding generated for five expressions

As expected, the runtimes do not significantly deviate from that for the non-encoded MBAs. Each encoded MBA has been simplified to the same expression as its simpler equivalent, which is of course encoded with the very same function.

4 Conclusion

The algorithm described in this paper is designed to simplify linear MBAs, which clearly is a restriction. Regarding the success at simplification of linear MBAs and the runtime taken, it is more than competitive with comparable tools.

MBA-Blast and MBA-Solver are efficient tools which are based on a groundbreaking transformation between $B = \{0, 1\}$ and B^n for any $n \in \mathbb{N}$ that dates back to 2007 [18]. In the publicly available implementation, MBA-Solver uses a lookup table in all cases and hence cannot be generalized to arbitrary variable counts.

We see that MBA-Solver's runtime increases drastically with four variables, but we cannot observe more of a trend since we cannot run it for higher numbers of variables. It uses an unhandy basis for bitwise expressions, and although that can, according to the paper [16], be changed, solving the equation system would cause much more effort then: While MBA-Blast uses `NumPy`'s `linalg.solve` function for solving it, MBA-Solver leverages the fact that all basis elements have exactly one 1 in their truth vectors and hence the equation system is trivially solved.

We summarize the main characteristics of SiMBA as follows:

1. It does not require any specific structure of input expressions as long as they can be rewritten as linear MBAs.
2. It checks whether input expressions are indeed linear.
3. It does not require any specific notation of the variables nor any declaration of them, but parses them automatically.
4. It aims for nicest solutions in all relevant cases for expressions with at most three variables. In more detail, it does so for expressions whose output depends on not more than three variables, i.e., after dropping unnecessary variables.
5. It is implemented to work for an arbitrary number of variables.
6. It can deal with arbitrarily high constants which are constantly reduced modulo 2^n for the word length $n \in \mathbb{N}$.
7. It does not use any `Python` package for simplification such as `SymPy` and no package such as `NumPy` for the solution of the linear equation system.
8. As our experiments suggest, it has a very good performance as compared to other existing tools.
9. It can handle linear MBAs whose outputs are encoded using affine functions.

10. It can be extended in a straightforward way to the simplification of polynomial and nonpolynomial MBAs.

We particularly want to point out that one main application of MBAs is given by *opaque predicates*, i.e., predicates which constantly evaluate to either 0 or 1, and SiMBA can easily simplify all linear MBAs used in opaque predicates to these constants.

As a major contribution, we proved that a linear MBA can be written as a linear combination of base bitwise expressions via a direct evaluation for tuples of zeros and ones and hence a transformation from B^n, for $n \in \mathbb{N}$, to $B = \{0, 1\}$ is not necessary — see Theorem 2.

5 Outlook to Nonlinear Mixed Boolean-Arithmetic Expressions

As mentioned previously, MBA-Blast and MBA-Solver are also applicable to polynomial MBAs. Just like those, SiMBA is extendable to this class of MBAs and to nonpolynomial MBAs as well after identifying all linear subexpressions of a nonlinear MBA. Unfortunately, it seems that a global straightforward approach via evaluation of nonlinear MBAs on a small set of input values is not possible.

Table 7 shows a comparison of an adaption of SiMBA with MBA-Solver for the simplification of linear, polynomial and nonpolynomial MBAs provided by MBA-Solver's Github repository [9]. While both tools can, in combination with Z3, verify all MBAs' equivalence to their corresponding simpler expressions, SiMBA simplifies all MBAs exactly equivalently as their simpler expressions. Additionally it simplifies 44 of 104 additional nonlinear MBAs which are marked unsolvable by MBA-Solver exactly as their simpler expressions, and it outperforms MBA-Solver regarding runtime in all categories. In contrast to MBA-Solver, SiMBA does not require any information whether an MBA is polynomial or even linear, and it does not use the additional hints about linear subexpressions provided for nonpolynomial MBAs.

Category	**Total**	**MBA-Solver**		**SiMBA**	
		Solved	**Runtime**	**Solved**	**Runtime**
Linear	1 008	1 008	0.01546 s	1 008	0.00250 s
Polynom.	1 008	1 008	0.02271 s	1 008	0.00326 s
Nonpolyn.	899	109	0.07965 s	899	0.00957 s

Table 7: Comparison of a simple adaption of SiMBA with MBA-Solver on the datasets of linear, polynomial and nonpolynomial MBAs provided by MBA-Solver's Github repository [9]

In contrast to the algorithm described in Section 3.2, SiMBA's adaption parses an input expression into an acyclic syntax tree (AST), identifies and simplifies linear subexpressions and applies simple additional tricks.

References

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

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

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

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

[5] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating MBA-based obfuscation. In *Proceedings of the 2nd ACM Workshop on International Workshop on Software PROtection (SPRO '16)*, pages 27–38, Vienna, Austria, October 2016. ACM.

[6] Matteo Favaro. NeuReduce. `https://github.com/fvrmatteo/NeuReduce`, 2020.

[7] 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 Association for Computational Linguistics 2020*, EMNLP 2020, pages 635–644. Association for Computational Linguistics, November 2020.

[8] UNH SoftSec Group. MBA-Blast code and dataset for USENIX Security '21. `https://github.com/softsec-unh/MBA-Blast`, 2020.

[9] UNH SoftSec Group. MBA-Solver code and dataset. `https://github.com/softsec-unh/MBA-Solver`, 2021.

[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, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. MBA-Blast. Unveiling and simplifying mixed Boolean-arithmetic obfuscation. In *Proceedings of the 30th USENIX Security Symposium*, USENIX 2021, pages 1701–1718. USENIX Association, August 2021.

[12] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. AI-based blackbox code deobfuscation. Understand, improve and mitigate. February 2021.

[13] Benjamin Reichenwallner and Peter Meerwald-Stadler. SiMBA code and dataset. `https://github.com/DenuvoSoftwareSolutions/SiMBA`, 2022.

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

[15] 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*, USENIX 2022, pages 3055–3073, Boston, MA, August 2022. USENIX Association.

[16] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, and Qiaoyan Yu. 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 2021, pages 651–664, Virtual, Canada, June 2021. Association for Computing Machinery.

[17] Yongxin Zhou and Alec Main. Diversity via code transformations. A solution for NGNA renewable security. In *The NCTA Technical Papers 2006*, pages 173–182, Atlanta, 2006. The National Cable and Telecommunications Association Show.

[18] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson. Information hiding in software with mixed Boolean-arithmetic transforms. In *Proceedings of the 8th International Conference on Information Security Applications*, WISA 2007, pages 61–75, Berlin, Heidelberg, August 2007. Springer-Verlag.