SaMBA: Increasing Mixed Boolean-Arithmetic Complexity Through Equality Saturation

Caroline König
University of Vienna
Faculty of Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
caroline.koenig@univie.ac.at

David Schmidt
University of Vienna
Faculty of Computer Science
Doctoral School Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
d.schmidt@univie.ac.at

Philip König
University of Vienna
Faculty of Computer Science
Doctoral School Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
philip.koenig@univie.ac.at

Patrick Felbauer
University of Vienna
Faculty of Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
patrick.felbauer@univie.ac.at

Sebastian Schrittwieser
University of Vienna
Faculty of Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
sebastian.schrittwieser@univie.ac.at

Abstract

Software protection often relies on code obfuscation to make tampering and reverse engineering time-consuming. A common technique is Mixed Boolean-Arithmetic (MBA) obfuscation, where simple expressions are rewritten by combining arithmetic with Boolean operations. This produces expressions that are functionally equivalent to the original but structurally complex and difficult to simplify. Conventional MBA generation typically relies on fixed templates, but prior work has shown that such patterns can often be simplified effectively.

We present *SaMBA*, a novel MBA generation approach based on equality saturation and e-graphs. E-graphs compactly represent many equivalent forms of an expression, and equality saturation applies rewrite rules until this space is fully explored. Guided by a negative-size cost function, *SaMBA* extracts the most complex equivalent, yielding semantically correct yet structurally diverse expressions.

We evaluate *SaMBA* against seven state-of-the-art MBA deobfuscation tools. Our results show that these approaches either fail to recover the original semantics or return expressions that remain syntactically complex, demonstrating *SaMBA*'s resilience against simplification attacks. Finally, integration with the Tigress obfuscator shows practical applicability, with only moderate performance overhead.

CCS Concepts

• **Security and privacy → Software and application security**; **Software security engineering**;

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASIA CCS '26, Bangalore, India
© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2356-8/2026/06
https://doi.org/10.1145/3779208.3785382

Keywords

Code Obfuscation, Mixed-Boolean Arithmetic, Equality Saturation, E-Graphs, Deobfuscation.

ACM Reference Format:
Caroline König, David Schmidt, Philip König, Patrick Felbauer, and Sebastian Schrittwieser. 2026. *SaMBA*: Increasing Mixed Boolean-Arithmetic Complexity Through Equality Saturation. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 01–05, 2026, Bangalore, India.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3779208.3785382

1 Introduction

Code obfuscation refers to a set of techniques used to protect programs from reverse engineering, tampering, and intellectual property theft. The goal of obfuscation is to transform source or binary code into a functionally equivalent but syntactically or structurally complex form that is difficult for human analysts or automated analysis systems to understand. Among the numerous obfuscation strategies, *Mixed Boolean-Arithmetic* (MBA) transformations have gained significant popularity in both research and industry due to their ability to produce expressions that are particularly hard to simplify or reverse.

MBA expressions combine arithmetic operations (e.g., addition, multiplication) with Boolean logic (e.g., AND, OR) to create expressions that are semantically equivalent to simpler forms but syntactically more complex. For example, a simple addition such as $x+y$ may be transformed into a deeply nested expression involving XORs and bitwise operations as depicted in Table 1. These transformations exploit algebraic identities, distributive properties, and arithmetic-Boolean interactions to make the original expression more complex. Due to their algebraic diversity and potential for deep nesting, MBA-based obfuscation is particularly resistant to conventional symbolic simplification and static analysis. A common application of MBA expressions is the construction of *opaque predicates*, which always evaluate to a known Boolean value but are designed to conceal this outcome. Such predicates introduce misleading branches into the

control flow, making it difficult to statically determine which code paths are actually executable.

In the ongoing arms race between attackers and defenders, a number of deobfuscation strategies for MBA have been proposed as well. For example, *MBA-Blast* [20] applies rewrite-based simplification and normalization to deobfuscate expressions. *Syntia* [1], leverage program synthesis guided by symbolic execution and input-output modeling, and *ProMBA* [16] integrates synthesis, linear algebra, and dynamic term rewriting. While these methods succeed in simplifying specific MBA patterns, we are able to demonstrate that their effectiveness diminishes substantially when faced with randomized, deeply nested, or structurally novel expressions.

In this paper, we propose a novel MBA generation approach based on equality saturation that is highly resilient to deobfuscation by exploiting the limitations of deobfuscation approaches when facing randomized, deeply nested, or structurally novel expressions. Equality saturation is a rewriting technique that systematically explores a large space of semantically equivalent expressions using equality graphs (e-graphs) [22]. Our approach, called *SaMBA*, repeatedly applies a set of algebraic, Boolean, and hybrid rewrite rules, guided by a negative-size cost function designed to maximize structural complexity while preserving semantics.

We demonstrate that expressions generated through our obfuscation approach *SaMBA* are highly resilient to deobfuscation attempts. Empirical evaluation against *Syntia*, *MBA-Blast*, and *SSPAM* shows that their approaches fail to recover the original semantics of our transformed expressions. Further, we show that these failures result from limitations in recursive normalization, structural pattern matching, and solver scalability. Additionally, practical usability is demonstrated by integrating *SaMBA* generated expressions in real-world code via the state-of-the-art obfuscator Tigress [2].

In summary, we make the following contributions:

(1) We present a novel methodology for generating structurally complex yet semantically equivalent MBA expressions using equality saturation and e-graph-based term rewriting;
(2) We introduce a novel negative-size cost function to guide the extraction of maximally complex expressions from equivalence graphs, favoring deep and nontrivial syntactic structures while preserving semantics;
(3) We construct a comprehensive set of arithmetic, bitwise, and algebraic rewrite rules (including commutativity, associativity, XOR decomposition, and hybrid rules) enabling diverse and non-canonical term-rewriting paths;
(4) We demonstrate the resilience of *SaMBA* by evaluating its output against seven state-of-the-art deobfuscation tools such as *Syntia* and *MBA-Blast*, identifying key failure types related to nested structures and pattern matching limitations.

To foster future work and reproducibility, we publish the source code of *SaMBA* at https://github.com/CDL-AsTra/SaMBA.

The remainder of this paper is structured as follows: Section 2 discussed preliminaries and introduces related work on MBA obfuscation and deobfuscation. Section 3 details our methodology. We present our evaluation of deobfuscation resilience and performance in Section 4. Finally, Section 6 draws conclusions.

2 Related Work and Preliminaries

In this section, we introduce MBAs and explain equivalence graphs (e-graphs) together with equality saturation, which form the preliminaries of our MBA generation methodology. We then review existing deobfuscation methods and analyze their approaches to handling MBA transformations. Finally, we introduce two complexity metrics that quantify the complexity of MBA expressions.

2.1 MBA Generation

Mixed-Boolean-Arithmetic (MBA) expressions combine arithmetic operations (e.g., addition, multiplication) with Boolean operators (e.g., AND, OR) to create obfuscated expressions. Zhou et al. [35] showed that such expressions can be systematically transformed within Boolean-arithmetic algebras (BA$[n]$), thereby increasing structural complexity while preserving semantic equivalence.

The process of generating an MBA relies on two components [8]:

- **MBA rewriting** denotes the practice of rewriting an operator with an equivalent expression as demonstrated in Table 1.
  This includes deciding on a set of rewrite rules that can be applied to an expression. Table 2 presents a set of rewrite rules that were utilized in our obfuscation approach.
- **Insertion of identities** effectively hides an expression e within invertible functions by rewriting e as $f(f^{-1}(e))$ on $\mathbb{Z}/2^n\mathbb{Z}$.

Furthermore, Zhou et al. [35] demonstrated that any operation can be rewritten into a more complex MBA form. They define *linear expressions* as polynomial MBA expressions over a Boolean-arithmetic algebra BA$[n]$ of dimension n with degree 1. The formal definition of a polynomial MBA expression is given as follows:

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*

$$e : (B^n)^t \rightarrow B^n$$

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 $j \in J_i$ and $i \in I$.

Linear MBAs represent a special case of polynomial MBAs in which each term contains exactly one bitwise expression and, optionally, a 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*

$$e : (B^n)^t \rightarrow B^n$$

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 $i \in I$.

Original Exp.	Obfuscated Exp.
$x + y$	$\Big(\Big(y\,\hat{}\,\big(\sim(\sim x) + 1\big) + \Big(2 * \big(y \,\&\, (\sim(\sim x) + 1)\big)\Big)\Big)\,\hat{}\,\big(\sim(1) + 1\big)\Big) + \Big(2 * \Big(\Big(y\,\hat{}\,\big(\sim(\sim x) + 1\big) + \Big(2 * \big(y \,\&\, (\sim(\sim x) + 1)\big)\Big)\Big) \,\&\, \big(\sim(1) + 1\big)\Big)$

Table 1: Comparison of original and obfuscated expressions

Furthermore, they proved that MBA deobfuscation is NP-hard, which implies that no general deterministic algorithm can efficiently simplify all MBA expressions. As a result, MBA obfuscation techniques gained significant popularity within the software protection community, both in academic research [19, 2, 3, 18] and in industry [13].

2.2 E-graphs

E-graphs [22] are data structures that compactly represent sets of expressions together with their equivalence classes under a congruence relation. Researchers originally developed e-graphs for theorem provers [6, 7], and later applied them for optimization tasks [29, 30, 14].

Formally, an e-graph is defined as a tuple (U, M, H) where a union-find data structure U stores an equivalence relation, the e-class map M maps e-class ids to e-classes, and a hash-cons table H maps from e-nodes to e-class ids, as in accordance with the definition given in [32]:

DEFINITION 3. *Let function symbols be denoted by f, g, and e-class identifiers (opaque identifiers) by a, b. The syntax of terms, e-nodes, and e-classes is defined as follows:*

$$
\begin{aligned}
\textit{Terms} \quad & t ::= f \mid f(t_1, \ldots, t_m), \quad m \geq 1 \\
\textit{E-nodes} \quad & n ::= f \mid f(a_1, \ldots, a_m), \quad m \geq 1 \\
\textit{E-classes} \quad & c ::= \{n_1, \ldots, n_m\}, \quad m \geq 1
\end{aligned}
$$

An e-graph *is a data structure representing equivalence relations among terms in a term language. Formally, an e-graph is a tuple (U, M, H) where:*

- *U is a union-find data structure [28] that maintains an equivalence relation over e-class identifiers (denoted $a \equiv_{id} b$).*
- *M is a mapping from e-class identifiers to e-classes. All equivalent e-class identifiers map to the same e-class:*

  $$a \equiv_{id} b \iff M[a] = M[b]$$

  An e-class identifier a refers to the e-class $M[\mathit{find}(a)]$.
- *H is a* hashconsing *map from e-nodes to canonical e-class identifiers. That is, $H : n \mapsto a$.*

2.3 Equality Saturation

Tate et al. [29] introduced equality saturation as a compiler optimization technique. Instead of applying a sequence of destructive transformations that risk converging to local optima, equality saturation enriches the intermediate representation (IR) with equality information. The IR must support the representation of semantically equivalent structures, for example, axioms such as

$$a * (b + c) = a * b + a * c \tag{1}$$

in more complex forms that also include loops and branches. To support this, Tate et al. [29] proposed *Program Expression Graphs* (PEGs), a specialized version of e-graphs that follows the notational conventions of Nelson et al. [22]. Since a PEG captures only a single expression of the input program, they later extended the concept to *E-PEGs*, which group equal PEG nodes into equivalence classes. The process of repeatedly inserting equality information until no further equalities can be introduced or a predefined threshold is reached is called *saturation*. A cost function then guides the extraction of the final representation by selecting the most suitable expression among the saturated equivalence classes.

Willsey et al. [32] expanded on this concept with the introduction of egg, an open-source, reusable e-graph library that enables efficient and widely applicable equality saturation. Their system introduced new optimizations for e-graph maintenance and demonstrated the applicability of equality saturation beyond traditional compilers, for example, in compiler super-optimizations [34]. They achieved this by developing an improved e-graph maintenance algorithm, which accelerates equality saturation by periodically rebuilding the e-graph while deferring maintenance tasks.

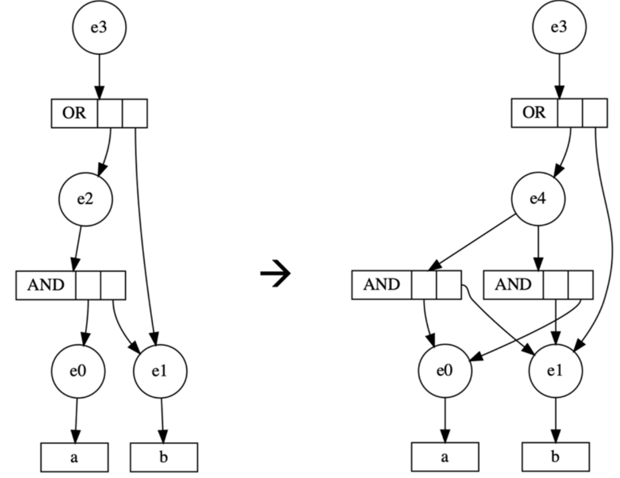


Figure 1: Example for building an e-graph with one rewriting starting with the expression $(a \,\&\, b) \mid b$, by applying the commutative rule.

2.4 MBA Deobfuscation

Proposed MBA deobfuscation techniques include term rewriting [9], stochastic program synthesis [1, 5, 21], symbolic simplification with

Boolean expression solvers [12], algebraic approaches [20, 24, 25, 33], and hybrid strategies that combine program synthesis with term rewriting [16].

2.4.1 Program Synthesis. *Syntia* [1] is a general code deobfuscation framework proposed by Blazytko et al., which aims at learning the semantics of VM-based instruction handlers by automatically recovering them for each handler through program synthesis. It uses Monte Carlo Tree Search (MCTS) to infer the semantics of obfuscated code from I/O samples and produces a deobfuscated representation. This approach has also been applied to MBA expressions, where Syntia successfully synthesized almost the entire dataset.

Other approaches, such as *QSynth* [5] and *Xyntia* [21], also rely on program synthesis for deobfuscation. *QSynth* concentrates on satisfying I/O constraints, whereas *Xyntia* emphasizes the compositional synthesis of higher-level constructs. Syntia, by contrast, uses MCTS to explore large program spaces efficiently, making it more general and robust, particularly effective for MBA expressions.

2.4.2 Algebraic Simplification. Liu et al. [20] demonstrated that MBA transformations exhibit identical behavior on 1-bit variables and arbitrary-length integers. Based on this insight, they introduced *MBA-Blast*, an approach that simplifies MBA expressions by reducing them within the 1-bit space. The method transforms bitwise expressions into corresponding MBA expressions in 1-bit space to enable arithmetic reduction.

MBA-Blast was evaluated on two datasets: the first contained 62 MBA expressions collected from existing tools, while the second comprised approximately 10,000 MBA expressions with varying bit-lengths (8 to 64 bits), numbers of variables (1 to 10), and term counts (3 to 80). Moreover, *MBA-Blast* successfully simplified recursively obfuscated expressions generated by Tigress [2], a popularly used obfuscation framework for C code.

Another approach by Xu et al. [33] utilizes a parser to translate expressions into Abstract Syntax Trees (ASTs). These are then simplified by replacing linear sub-expressions using signature vectors that encode their semantics. The system then performs arithmetic reduction and small optimizations, producing an equivalent but simplified MBA expression.

Arybo [12] introduces symbolic manipulation techniques for simplifying and identifying MBA expressions at the bit level. *Arybo* models MBA expressions using bit-vector algebra and operates on the Algebraic Normal Form over the field $\mathbb{F}_2$. Its main contribution lies in providing a canonical symbolic representation of expressions that combine arithmetic and Boolean operators, which enables equivalence checking, simplification, and reverse engineering. The key idea is to translate MBA expressions into a bitwise symbolic form that relies solely on XOR and AND operators. This representation is both expressive and canonical, ensuring that semantically equivalent expressions always map to the same form.

SiMBA [24] is a deobfuscator designed for linear MBAs with an arbitrary number of variables. It transforms a given MBA into a linear combination of base bitwise expressions. Since the algorithm evaluates expressions over all combinations of zeros and ones, it is independent of the input representation and can also simplify non-linear MBA expressions, provided that they reduce to a linear form. *GAMBA* [25] extends this approach by iteratively identifying linear sub-expressions and simplifying them with *SiMBA*. Furthermore, it introduces factorization of non-linear sums and substitution of sub-expressions with variables to enable further reduction.

Rule Name	**Original (LHS)**	**Rewritten (RHS)**
	Commutative Properties	
commute-mul	$(x * y)$	$(y * x)$
commute-add	$(x + y)$	$(y + x)$
commute-and	$(x \& y)$	$(y \& x)$
commute-xor	$(x \hat{} y)$	$(y \hat{} x)$
commute-or	$(x \mid y)$	$(y \mid x)$
	Associative Properties	
assoc-mul	$(x * (y * z))$	$((x * y) * z)$
assoc-add	$(x + (y + z))$	$((x + y) + z)$
assoc-and	$(x \& (y \& z))$	$((x \& y) \& z)$
assoc-xor	$(x \hat{} (y \hat{} z))$	$((x \hat{} y) \hat{} z)$
assoc-or	$(x \mid (y \mid z))$	$((x \mid y) \mid z)$
	Boolean & Arithmetic Transformations	
not-mul	$(\sim(x * y))$	$(((\sim x) * y) + (-y - 1))$
not-add	$(\sim(x + y))$	$((\sim x) + ((\sim y) + 1))$
not-sub	$(\sim(x - y))$	$((\sim x) - ((\sim y) + 1))$
not-and	$(\sim(x \& y))$	$((\sim x) \mid ((\sim y)))$
not-xor	$(\sim(x \hat{} y))$	$((x \& y) \mid (\sim(x \mid y)))$
not-or	$(\sim(x \mid y))$	$((\sim x) \& (\sim y))$
neg-mul	$(-(x * y))$	$((-x) * y)$
sub-to-add	$(x - y)$	$(x + (-y))$
neg-to-not	$(-x)$	$((\sim x) + 1)$
	Distribution Rules	
distribute-mul	$(x * (y + z))$	$((x * y) + (x * z))$
distribute-sub	$(x * (y - z))$	$((x * y) - (x * z))$
factor-mul	$((x * y) + y)$	$((x + 1) * y)$
double-add	$(2 * x)$	$(x + x)$
neg-distribute	$(-(x + y))$	$((-x) + (-y))$
	XOR and AND/OR Mix Rules	
add-to-xor	$(x + y)$	$((x \hat{} y) + (2 * (x \& y)))$
xor-rewrite	$(x \hat{} y)$	$((x \mid y) - (x \& y))$
add-to-and-or	$(x + y)$	$((x \& y) + (x \mid y))$
sub-to-xor	$(x - y)$	$((x \hat{} (-y)) + (2 * (x \& (-y))))$
and-to-add	$(x \& y)$	$(((1 * x) + (1 * y)) - (x \mid y))$
or-to-add	$(x \mid y)$	$((x + y) + 1 + ((\sim x) \mid (\sim y)))$
add-via-xor	$(x + y)$	$((2 * (x \mid y)) - (x \hat{} y))$

Table 2: Excerpt of utilized rewrite rules used for MBA expression transformations in *SaMBA*.

2.4.3 Pattern Matching. Eyrolles et al. [9] proposed an approach based on pattern matching, called *SSPAM*. *SSPAM* applies invertible rewrite rules to simplify MBA expressions at the word level, thereby avoiding expensive bit-blasting or SAT-based procedures. *SSPAM* represents expressions as ASTs or term graphs and combines pattern matching with arithmetic simplification to reduce complexity. The patterns rely on known MBA equivalences and are applied in

reverse to collapse obfuscated expressions. To increase robustness, *SSPAM* integrates a flexible pattern matcher that employs SMT solvers such as Z3 to verify semantic equivalence in cases where constant folding or compilation artifacts introduce variations.

2.4.4 Hybrid Approach. *ProMBA* [16] combines program synthesis, term rewriting, and linear algebraic simplification in an iterative approach that extends beyond pure synthesis. The method learns rewrite rules dynamically and applies them to simplify obfuscated code. It first targets linear sub-expressions for simplification, then employs a divide-and-conquer strategy to synthesize equivalent counterparts of selected sub-expressions, thereby reducing the search space. Each discovered equivalence is integrated into the rewrite rule set, which incrementally improves the minimization process. The authors evaluated *ProMBA* on more than 4,000 non-linear MBA expressions. The results show that it significantly outperforms existing deobfuscation approaches such as *MBASolver* and *Syntia* with respect to simplification success rate and expression size reduction. Correctness is guaranteed by using the Z3 SMT solver [6] to verify all generated rewrite rules.

2.4.5 Equality Saturation. Lee et al. [17] proposed a novel MBA deobfuscation approach called *EMBA*, which leverages equality saturation. Compared to *ProMBA* and *GAMBA*, *EMBA* achieves comparable success rates while being less time-consuming. However, it fails to deobfuscate expressions generated by Code Virtualizer [23].

2.5 Complexity Metrics

To evaluate the complexity of both the obfuscated expressions and the corresponding deobfuscated results, we use two quantitative metrics: (1) the size of the *directed acyclic graph* (DAG) and (2) the degree of MBA alternation.

2.5.1 DAG Size. The definition of a DAG with regards to an expression is as follows [8]:

Definition 4. *The DAG representation of an MBA expression is an acyclic graph G where:*

1. *all leaves represent constant numbers or variables, other nodes represent arithmetic or bitwise operators;*
2. *an edge from a node v to a node v' means v' is an operand of v;*
3. *there is only one root node;*
4. *common expressions are shared, which means they only appear once in the graph.*

Following Eyrolles [8], we define the size of an expression with respect to its DAG representation as the total number of nodes, including operators, variables, and constants. Since subexpressions occur only once in a DAG, they contribute only once to the overall size of the expression.

2.5.2 MBA Alternation. Since MBA expressions combine arithmetic and Boolean operators, MBA *alternation* captures how often an expression switches between these operator classes. The formal definition proposed by Eyrolles [8] is as follows:

Definition 5. *For a graph G = (V, E) with V the set of vertices and E the set of edges, the MBA alternation $alt_{MBA}(G)$ is:*

$$alt_{MBA}(G) = |(v1, v2) \text{ such that } type(v1) \neq type(v2)|,$$

where (v1, v2) ∈ E represents the edge linking the two vertices v1, v2 ∈ V.

Algorithm 1: MBA Obfuscation via Equality Saturation

```
1:  Function NegSizeCost(node):
2:      childCosts ← Σ_{c∈children(node)} NegSizeCost(c)
3:      return childCosts − 1    // Negative to maximize size
4:  Function ExtractLargestExpr(egraph, root):
5:      extractor ← Extractor(egraph, NegSizeCost)
6:      return extractor.findBest(root)
7:  Define set of rewrite rules R including:
8:      Commutative: (x + y) ⇒ (y + x)
9:      Associative: (x + (y + z)) ⇒ ((x + y) + z)
10:     Boolean: (~(x & y)) ⇒ ((~x) | (~y))
11:     Arithmetic: (x − y) ⇒ (x + (−y))
12:     Mixed: (x + y) ⇒ ((x^y) + (2 ∗ (x & y)))
13: Function ObfuscateExpression(expr, iterations, nodeLimit):
14:     currentExpr ← expr
15:     for i = 1 to iterations do
16:         runner ← Runner().withExpr(currentExpr).withNodeLimit(nodeLimit)
17:         runner.run(R)    // Apply all rewrite rules
18:         egraph ← runner.egraph
19:         root ← runner.roots[0]
20:         currentExpr ← ExtractLargestExpr(egraph, root)
21:     end for
22:     return currentExpr
```

3 Methodology

In this section, we introduce *SaMBA*, our approach for generating obfuscated MBA expressions. We further present our methodology for measuring deobfuscation resilience, describe the integration of *SaMBA* into Tigress [2] to enable automated obfuscation of real-world code, and outline our benchmarking.

3.1 SaMBA

In *SaMBA*, we employ equality saturation to generate structurally complex yet semantically equivalent MBA expressions. We iteratively apply a curated set of rewrite rules and utilize a cost function that prioritizes larger abstract syntax trees to identify the global rewriting optimum. Each iteration saturates the e-graph with equivalent forms and extracts the most complex representation.

In Algorithm 1, we provide an overview of *SaMBA*'s methodology for generating MBA expressions. For constructing e-graphs and performing equality saturation, *SaMBA* builds on the egg library [32].

Cost Function. To generate increasingly complex MBA expressions, a cost function guides *SaMBA* towards larger, more complex expressions. The cost function returns the negative size of an

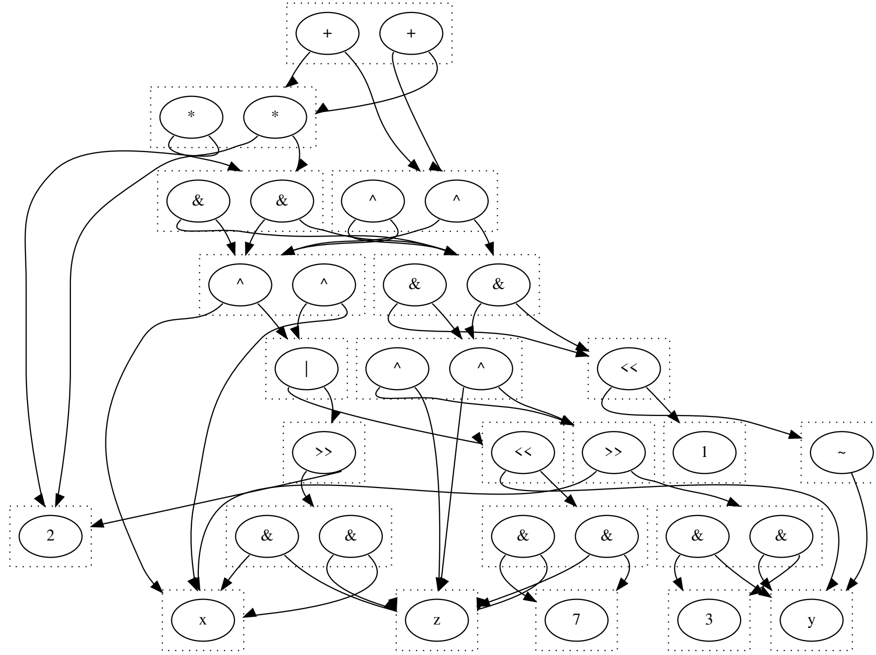


Figure 2: Final e-graph after equality saturation. E-classes are depicted by dashed boxes and contain equivalent e-nodes. Child e-classes are connect to their e-nodes via edges. New e-nodes and edges are added for each rewrite.

expression's AST. By minimizing this negative value, the extractor implicitly selects expressions with maximal depth and node count.

$$\text{cost} = \sum_{\text{children}} \text{cost}_{\text{child}} - 1 \tag{2}$$

This ensures that subtrees of larger size are preferred during extraction, guiding the rewrite engine toward maximal expansion.

Expression Extraction. To extract complex expressions from the e-graph, we employ the egg Extractor module together with our cost function. The extractor navigates the e-graph and selects the most complex equivalent of the input expression. This process enables the selection of structurally rich representations after each rewriting iteration. To mitigate uncontrolled growth of the e-graph, we can manually constrain the maximum number of nodes. Without this bound, the cost function would continuously favor the biggest possible subtree, potentially causing the e-graph to explode in size and leading to overflows.

We also defined a comprehensive set of 41 rewrite rules to encode arithmetic, boolean, and algebraic identities, partially originating from Henry Warren's *Hacker's Delight* [31]. These include commutative and associative transformations, bit-wise and arithmetic identities, distribution rules, and hybrid transformations such as rewriting XOR or AND/OR compositions. These rules, such as depicted in Table 2 or equations (3) and (4), were applied within a fixed-depth iteration loop.

$$(x + y) = (y + x) \quad \text{(Commutativity)} \tag{3}$$

$$(x + y) = ((x \hat{\ } y) + (2 * (x \,\&\, y))) \quad \text{(XOR Decomposition)} \tag{4}$$

Iterative Saturation. In each round, we parse an expression into an AST and run it through the equality saturation engine. The engine applies all applicable rewrite rules until either a node limit or an iteration threshold is reached. After saturation, the extractor retrieves the most complex expression available from the resulting e-graph. Figure 2 depicts our final e-graph used by *SaMBA*. This expression then serves as input for the next rewriting round. To monitor the transformation process, each iteration also produces a visual representation of the e-graph.

We execute the entire pipeline for a fixed number of iterations. The final output is an MBA expression that preserves semantic equivalence to the original input while exhibiting significantly increased structural complexity. This process produces obfuscated expressions that are hard to simplify, thereby strengthening their resilience against symbolic simplification and deobfuscation techniques, while remaining lightweight, see Section 4.

To ensure that every rewriting rule is formally correct, we verified them with the SMT solver Z3 [6] over 32-bit bitvector variables. For reproducibility, we publish the corresponding Python script in the artifact [15]. To further confirm that the resulting MBAs preserve semantic equivalence to the original expression, we evaluate both the obfuscated and the original forms over all possible input combinations for binary variables and compare their outputs.

Tigress Plugin. To enable the use of *SaMBA* for real-world code, we integrated with the state-of-the-art C source-to-source obfuscator Tigress [2] through its plugin infrastructure. A Tigress plugin for

MBA obfuscations is a C file that defines MBA rules for arithmetic and boolean operations, which Tigress applies during obfuscation.

Automatically generating plugins offers the advantage of producing distinct MBA rules for different binaries. In addition, the generated MBAs are not publicly accessible by reversing Tigress, which increases their resilience against deobfuscation.

3.2 Deobfuscation Resilience Measurements

To empirically evaluate the resilience of *SaMBA*'s generated MBAs, we analyze whether seven state-of-the-art MBA deobfuscators, selected based on the availability of their artifacts and approaches, are capable of simplifying obfuscated expressions (*GAMBA*, *Syntia*, *SSPAM*, *ProMBA*, *Arybo*, and *MBA Blast*).

We first performed an initial manual evaluation of all deobfuscation approaches to study the success of the process in detail. Specifically, we examined termination of the simplification procedure, semantic equivalence of the resulting expressions, and their complexity. After this initial evaluation, we conducted an automated assessment of functional deobfuscation approaches on a dataset of 1,000 unique expressions. For each expression, we applied *SaMBA*'s obfuscation and subsequently executed the deobfuscation approaches. To measure their success, we compared the complexity of the original expression, the obfuscated version, and the deobfuscated result.

3.2.1 Case Study Evaluation. To generate MBAs for our case study evaluation, we repeatedly applied a set of rewrite rules through equality saturation over five iterations. In each iteration, an inner loop invokes the rewriting procedure, which attempts to generate the largest equivalent sub-expression within a predefined node limit. We set the initial node limit to 55 within the e-graph to prevent timeouts in the deobfuscation tools and avoid overflow in *SaMBA*. This also ensures that deobfuscated expressions remain interpretable during manual analysis.

3.2.2 Batch Evaluation. For our batch evaluation, we used the dataset from a fork of NeuReduce [10, 11], which included 100,000 samples of obfuscated expressions and their ground truth. After eliminating duplicates, we extracted 1,000 unique expressions, which we then obfuscated using *SaMBA*, setting the node limit to 80 within the e-graph. If an overflow occurred, the node limit was halved. This adaptive mechanism continued until the node limit dropped below 10, at which point further expansion of the expression was skipped. After completing all iterations, we recorded the elapsed time and accumulated it over all processed expressions.

To automatically evaluate the complexity of expression, we use two approaches (1) DAG size and depth, and (2) arithmetic and bitwise alternation counts.

DAG Size and Depth. As approaches such as *GAMBA* maintain semantic equivalence, the complexity of the resulting expressions becomes a primary criterion for success. To assess the complexity of the ground truth, the obfuscated expression, and the deobfuscation by the results DAG depth as well as size were utilized.

For this, the MBA expressions were parsed to construct a corresponding DAG tree. To analyze the complexity of MBA expressions, the DAG structure is constructed with two main classes. The Node class represents individual components of an expression, such as variables, constants, and operators. The ExpressionDAG class manages these nodes, ensuring reuse of identical subexpressions to minimize redundancy and optimize resource usage.

The DAG's depth is computed using memoized recursion, where previously computed depths are stored in a dictionary. This avoids recalculating depths for nodes encountered multiple times, which is particularly important in DAGs where nodes may share substructures. The computed depth represents the longest path from the root to any leaf node, while the size of the DAG is the total number of nodes it contains. These metrics are then averaged across ground truth, obfuscated, and deobfuscated expressions to evaluate their structural complexity. This approach ensures efficient representation of expressions and provides meaningful insights into how obfuscation and deobfuscation affect their complexity.

Alternation. Additionally, effective obfuscation techniques often increase complexity through frequent transitions between arithmetic and bitwise operations. The implementation classifies operators into two distinct categories: arithmetic operations (addition, subtraction, multiplication, division, modulo, floor division, power) and bitwise operations (AND, OR, XOR, shifts). It then constructs and traverses the AST to count transitions between these categories.

The core analysis employs a recursive approach through the analyze function, which tracks both the current operation type and accumulates alternation counts when adjacent operations switch between categories. By comparing alternation metrics, we can evaluate both the potency of obfuscation techniques and the efficacy of deobfuscation.

3.3 Performance Measurements

Additionally, the integration into Tigress allows us to benchmark the execution time of unobfuscated code against code obfuscated with *SaMBA*. Understanding the runtime overhead introduced by our transformations is crucial, since a significant performance degradation could render the obfuscated program unusable and therefore make the obfuscation practically ineffective.

To evaluate the performance impact, we obfuscated `sha512sum` from the GNU Core Utilities. We selected `sha512sum` due to the numerous computations it performs during regular execution. To assess the overhead of the obfuscation, we created a benchmark dataset containing seven randomly generated files ranging from 1KB to 1GB, and 107 binaries from coreutils, each smaller than 1MB. Additionally, we included a larger real-world example in the form of a Linux kernel archive file (220MB). For each test case, we executed both the obfuscated and unobfuscated `sha512sum` binaries and compared their execution times.

4 Evaluation

Collberg et al. [4] proposed measuring obfuscation strength based on its potency and resilience. While potency refers to the confusion a transformation adds to the code and to what degree it complicates code understanding, resilience measures how difficult it is to undo a transformation. In our evaluation, we focus on resilience: we measure how effectively existing deobfuscators can simplify MBAs generated by our approach.

We applied different node limits depending on the evaluation setting. For our case study evaluation, we restricted e-graphs to 55

nodes, since higher limits caused several tools to time out. In our automated batch evaluation, we raised the limit to 80 nodes. Since our algorithm halves the node size when an overflow occurs, the higher limit ensures that the resulting MBA expressions remain sufficiently large and complex even after such reductions. Across both settings, outputs that remained syntactically complex despite semantic correctness were classified as failures.

The average obfuscation processing time per expression for all five rounds of rewriting was approximately 12 milliseconds on a Core i5-1135G7 for up to 250 nodes.

4.1 Case Study Evaluation

In the following, we discuss deobfuscation failures we observed with specific tools in detail. The selected cases are representative of structural limitations in current deobfuscation approaches, and thus the observations extend beyond the particular examples presented.

4.1.1 *Stochastic Program Synthesis.*

Syntia. Even after the first out of five rounds of consecutive obfuscation, *Syntia* was not able to recover the original semantics of the ground truth

$$2 * ((x * y) + z) \tag{5}$$

and instead producing the expression

$$(((x * y) - y) + z) \tag{6}$$

which failed equivalence testing across the full input space. This behavior reflects a core characteristic of *Syntia*'s architecture. Rather than structurally reversing an expression through deterministic symbolic simplification or algebraic analysis, *Syntia* relies on symbolic execution combined with MCTS to iteratively synthesize a semantically equivalent candidate expression. The synthesis process attempts to maximize semantic fidelity based on sampled input-output behavior, guided by a reward function over symbolic expressions.

Additionally, as stochastic synthesis methods rely on random mutations and evaluations, these methods often struggle to decompose or simplify nested or non-linear sub-expressions. An expression that collapses into a basic expression, such as $x + y$, could therefore be easily deobfuscated, but as the complexity increases, the deobfuscation repeatedly fails. *Syntia* can therefore not guarantee that the resulting expression is semantically equivalent. These are also known limitations as described by Blazytko et al. [1].

ProMBA. This deobfuscator combines syntax-guided program synthesis with term rewriting and linear-MBA simplification in an iterative deobfuscation loop. Given an MBA expression, it first applies an off-the-shelf linear-MBA solver to collapse any degree-1 sub-terms, then selects a non-linear sub-expression and poses a SyGuS synthesis query to find a smaller, semantically equivalent replacement. Each successful synthesis step yields a new rewrite rule, which *ProMBA* adds to its rule set before repeating the process, alternating between solver-driven simplification and learned rewrite applications until the entire expression has been simplified.

As *ProMBA* checks equivalences while simplifying expressions, it guarantees that its result is at least semantically the same as the obfuscated expression. Nonetheless, *ProMBA* suffers from several practical shortcomings when applied to complex MBAs, which have been described by Lee et al. [16]. First, its reliance on solver-based enumeration means that, as the size or nesting depth of the input expression increases, individual synthesis subproblems can consume a large amount of time before reaching *ProMBA*'s internal timeout or node-count limit. Second, even successful runs do not guarantee recovery of the original, human-readable expression. Instead, *ProMBA* returns some semantically equivalent expression, which may itself remain a deeply complex MBA expression rather than a clear arithmetic identity, such as:

$$\begin{aligned}
&(((\sim x) \,\&\, (((y \,\&\, z) * (y \mid z)) + ((y \,\&\, (\sim z)) * (z \,\&\, (\sim y)))))\\
&\quad * ((\sim x) \mid (((y \,\&\, z) * (y \mid z)) + ((y \,\&\, (\sim z)) * (z \,\&\, (\sim y)))))\\
&+ (((\sim x) \,\&\, (\sim(((y \,\&\, z) * (y \mid z)) + ((y \,\&\, (\sim z)) * (z \,\&\, (\sim y))))))\\
&\quad * ((((y \,\&\, z) * (y \mid z)) + ((y \,\&\, (\sim z)) * (z \,\&\, (\sim y)))) \,\&\, x))
\end{aligned} \tag{7}$$

with the ground truth being

$$\sim x * (y * z) \tag{8}$$

The deobfuscated expression shows a substantial increase in complexity compared to its ground truth equivalent. The DAG size expands from 5 nodes to 21 nodes, representing a 4.20 increase in expression components. Similarly, the DAG depth extends from 2 levels to 8 levels, marking an increase of 4 in computational path length. The obfuscation holds as the final expression shows a quadrupled increase in complexity compared to the ground truth.

QSynth and Xyntia. As both *QSynth*'s and *Xyntia*'s core algorithms are similar to *Syntia*, which is the synthesis from random input-output examples, we did not further evaluate them analogous to Lee et al. [16].

4.1.2 *Algebraic Simplification.*

GAMBA. Similarly to *ProMBA*, *GAMBA* returns a semantically correct but syntactically complex MBA expression for the obfuscated ground truth $((y - \sim x) - 1)$.

$$\begin{aligned}
&\sim\!\big(y\hat{\ }(-(\sim x))\big) + \Big(2 \,\&\, \big((2 \,\&\, y \,\&\, (-(\sim x))) * (2 \mid (y \,\&\, (-(\sim x))))\\
&\qquad + (2 \,\&\, \sim(y \,\&\, (-(\sim x)))) * (-3 \,\&\, y \,\&\, (-(\sim x))) + y\hat{\ }(-(\sim x))\big)\Big)\\
&\quad * \Big(2 \mid \big((2 \,\&\, y \,\&\, (-(\sim x))) * (2 \mid (y \,\&\, (-(\sim x))))\\
&\qquad + (2 \,\&\, \sim(y \,\&\, (-(\sim x)))) * (-3 \,\&\, y \,\&\, (-(\sim x))) + y\hat{\ }(-(\sim x))\big)\Big)\\
&\quad + \Big(2 \,\&\, \sim\big((2 \,\&\, y \,\&\, (-(\sim x))) * (2 \mid (y \,\&\, (-(\sim x))))\\
&\qquad + (2 \,\&\, \sim(y \,\&\, (-(\sim x)))) * (-3 \,\&\, y \,\&\, (-(\sim x))) + y\hat{\ }(-(\sim x))\big)\Big)\\
&\quad * \Big(-3 \,\&\, \big((2 \,\&\, y \,\&\, (-(\sim x))) * (2 \mid (y \,\&\, (-(\sim x))))\\
&\qquad + (2 \,\&\, \sim(y \,\&\, (-(\sim x)))) * (-3 \,\&\, y \,\&\, (-(\sim x))) + y\hat{\ }(-(\sim x))\big)\Big)\\
&\quad - (2 \,\&\, y \,\&\, (-(\sim x))) * (2 \mid (y \,\&\, (-(\sim x))))\\
&\qquad - (2 \,\&\, \sim(y \,\&\, (-(\sim x)))) * (-3 \,\&\, y \,\&\, (-(\sim x)))
\end{aligned} \tag{9}$$

Although equivalent to the originally obfuscated expression, it remains syntactically complex. The DAG size remains expanded from 6 nodes of the ground truth to 38 nodes, representing a 6.33 increase in expression components. DAG depth extended from 3 levels to 12 levels, marking a 4.00 increase in computational path length. Additionally, the deobfuscated expression also shows 9 alterations between arithmetic and bitwise operations, in contrast to the pure expression structure with zero MBA alternations of the ground truth.

Furthermore, one can force *GAMBA* to timeout by constructing a sufficiently large MBA expression, prohibiting any deobfuscation attempt. This is due to the fact that *GAMBA* effectively only hands linear fragments to the linear deobfuscator *SiMBA*, therefore leaving collapsed sub-terms in their non-linear context. In an experimental setup, when the ground truth is given, *GAMBA* is therefore capable of producing verifiable and semantically correct results, while remaining obfuscated syntactically. This is in line with the current implementation of *GAMBA* [25].

MBA-Blast. A fundamental limitation of *MBA-Blast* lies in its handling of nested or compositional expressions, as described by related work [20, 26]. The tool operates primarily through pattern matching and normalization at the top-level structure of expressions, which makes it effective for flat, canonical MBA identities such as

$$x + y = (x \hat{} y) + 2 * (x \,\&\, y) \tag{10}$$

In contrast, when obfuscation uses recursive rewrites or multi-layered encodings, where MBA identities are applied to sub-terms nested within larger constructs, *MBA-Blast* fails to simplify them effectively. This is because its matching engine lacks recursive normalization and does not perform symbolic analysis across subtrees of the expression. As a result, nested MBA constructs remain opaque to *MBA-Blast*, making the deobfuscation ineffective against obfuscation schemes that leverage hierarchical or deeply structured compositions.

In our case, *MBA-Blast* terminated without a result due to the tool's inability to correctly expand the parentheses even when only linear transformations have been applied. As this is an inherent flaw in the tool's structure, there was no way to actually deobfuscate any of the nested expressions we generated.

Schloegel et al. [26] adapted the code of *MBA-Blast* to allow for nested sub-terms if they are concatenated using the same operand within the expression. Additionally, the authors constrained their obfuscation approach significantly to allow for a deobfuscation by *MBA-Blast*, which overall did neither reflect the capabilities of the deobfuscator nor the full extent of the *Loki* obfuscations. Therefore, no adaptation of the *MBA-Blast* codebase has been performed.

MBA-Solver. MBA-Solver was not able to deobfuscate any nested expression, therefore failing similarly as *MBA-Blast* albeit due to the cases falling outside of the normalization range [33].

4.1.3 Pattern Matching.

SSPAM. While *SSPAM* provides an efficient method for deobfuscating MBA expressions using invertible rewrite rules and SMT-based equivalence checking, it demonstrates limitations when faced with deeply nested and structurally complex encodings [26]. In our case, *SSPAM* fails to correctly simplify an MBA-obfuscated expression that utilizes multiple layers of known identities. This expression includes symbolic encodings of the increment operation and obfuscated constants such as

$$(\sim 1 \hat{} 1) + 2 * (\sim 1 \,\&\, 1) \tag{11}$$

For example, when trying to deobfuscated the obfuscation result after one iteration of equality saturation for the expression $((y - \sim x) - 1)$, *SSPAM* produces the following result:

$$((((2 * x) + (2 * y)) + (((x + y) + 1) \hat{} 3)) + 2) \tag{12}$$

While this is a simplification, it is not semantically equivalent to the ground truth. This results from limitations in structural pattern matching, where the pattern matcher only operates at the current tree depth and does not recursively normalize inner sub-expressions as described by Eyrolles et al. [9]. This demonstrates a key weakness in *SSPAM*'s approach. Although effective against syntactically straightforward MBA encodings, it is vulnerable to randomized and layered constructions that deliberately obscure canonical algebraic forms and induce solver failure or misidentification. Additionally, although *SSPAM* is equipped with a predefined set of equivalence-preserving rewrite rules, its pattern matcher remains inherently limited in scope. In the public *SSPAM* implementation, the provided rule set does not recursively normalize constructs such as

$$(\sim(\sim x \hat{} 1)) + 2 * (\sim(\sim x \,\&\, 1)) \tag{13}$$

Consequently, that expression would not be simplified to

$$x + 1 \tag{14}$$

4.1.4 Bit Blasting.

Arybo. When tasked with simplifying the following 64-bit expression:

$$\begin{aligned} &(\sim x \,\&\, \big((y \,\&\, z) * (y \mid z) + (y \,\&\, \sim z) * (z \,\&\, \sim y)\big)) \\ &\quad * \big(\sim x \mid \big((y \,\&\, z) * (y \mid z) + (y \,\&\, \sim z) * (z \,\&\, \sim y)\big)\big) \\ &\quad + (\sim x \,\&\, \sim\big((y \,\&\, z) * (y \mid z) + (y \,\&\, \sim z) * (z \,\&\, \sim y)\big)) \\ &\quad * \big((y \,\&\, z) * (y \mid z) + (y \,\&\, \sim z) * (z \,\&\, \sim y)\big) \,\&\, \sim(\sim x) \end{aligned} \tag{15}$$

Arybo did not terminate within ten hours and therefore failed to deobfuscate the given expression. This has also been demonstrated by Schloegel et al. [26]. As *Arybo* transforms every operator into ANF over the field $\mathbb{F}_2$, then performing Gaussian elimination, this approach is incredibly fast for 1-bit expressions. A deeply nested, 64-bit expression lets the ANF expansion grow exponentially, effectively hindering any obfuscation attempt by exhausting memory and solver-time. Therefore, *Arybo* failed for all obfuscated expressions.

Takeaways

We identified the following deobfuscation limitations:

- Stochastic Synthesis (*Syntia*, *ProMBA*): For more complex MBAs, computational overhead exceeds available resources
- Algebraic Simplifiers (*GAMBA*, *MBA-Blast*): Produce semantically correct but syntactically complex results; fail on nested expressions
- Pattern Matching (*SSPAM*): Limited by non-recursive matching; vulnerable to layered constructions
- Bit Blasting (*Arybo*): Fails to terminate on 64-bit expressions due to exponential ANF expansion

Our approach effectively resists all tested deobfuscation techniques while maintaining semantic equivalence.

4.2 Batch Evaluation

We restricted our batch evaluation to *GAMBA*, *ProMBA*, and *Syntia*, since prior results demonstrated the limitations of other analyses, see Section 4.1. We applied each analysis to deobfuscate 1,000 unique obfuscated samples. For *GAMBA* and *ProMBA*, we measured the complexity of the resulting expressions using DAG size and depth, as both approaches guarantee semantic correctness and thus allow structural complexity to serve as the primary evaluation metric. In contrast, for *Syntia*, we focused on correctness, since it already failed to synthesize results for a substantial number of expressions in the case study.

4.2.1 GAMBA and ProMBA. Our analysis of deobfuscation effectiveness shows that both produce outputs with substantially larger sizes than the ground truth. We define the *Recovery rate* as

$$\text{Recovery rate} = \frac{\text{Obf} - \text{Deobf}}{\text{Obf} - \text{GT}} \times 100, \tag{16}$$

where Obf, Deobf, and GT denote the values (either size, DAG size, DAG depth, or alternation) of the obfuscated, deobfuscated, and ground-truth expressions. This metric quantifies the percentage of obfuscation-induced overhead that deobfuscation removes.

DAG Size. DAG size serves as a fundamental metric for expression complexity evaluation because it directly quantifies the computational resources required for expression evaluation. This metric correlates with both memory consumption and execution time in practical implementations. From a theoretical perspective, DAG size represents the minimum number of operations needed to compute an expression, making it a language-independent measure of inherent complexity. The substantial size disparities between deobfuscated and ground truth expressions demonstrate that current deobfuscation techniques, while partially effective at reducing complexity, still yield representations that remain significantly more complex, and therefore maintain both potency and resilience.

On average, *GAMBA* yields deobfuscated expressions that are more than three times larger than the original, while *ProMBA* produces results exceeding four times the original size. The reduction in complexity from obfuscated to deobfuscated expressions remained modest. On average, *GAMBA* achieved a recovery rate of 41.8%, while the rate of *ProMBA* was only 26.99%, see Table 3.

Method	Size (Obf.)	Size (GT)	Size (Deobf.)	Recovery Rate
GAMBA	56.19	10.21	36.97	41.80%
ProMBA	56.19	10.21	43.78	26.99%

Table 3: Comparison of average DAG sizes. Recovery quantifies the percentage of the gap between obfuscated code and ground truth that deobfuscation closes.

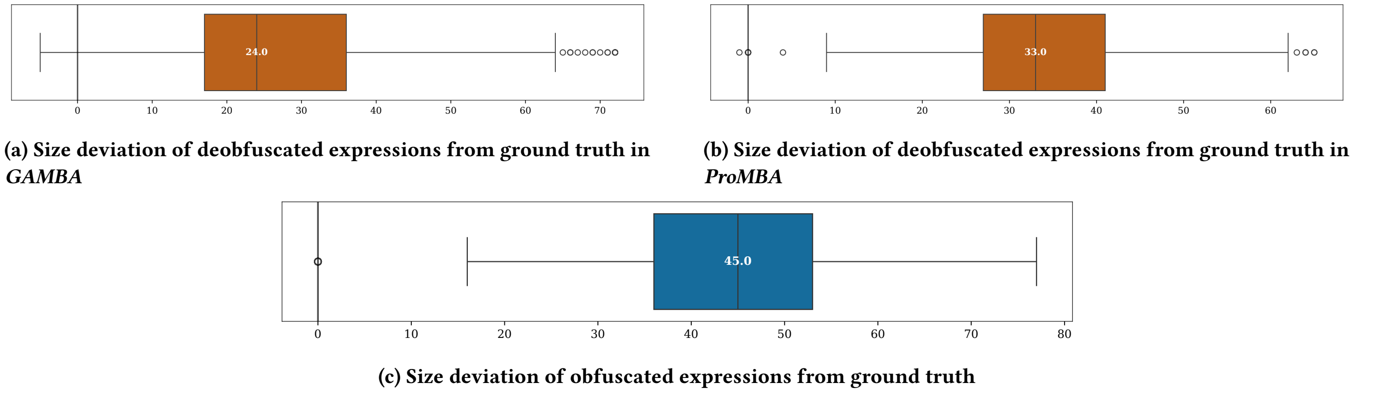
Overall, Figure 3c shows size deviation between expressions obfuscated with *SaMBA* and the ground truth. The interquartile range spans approximately 38 to 50 nodes with a median of 45. Whiskers extend from about 15 to 72 nodes, with an outlier near the 0 mark. *SaMBA* increases expression size consistently by a substantial amount, creating significantly more complex expressions. The narrower interquartile range suggests the obfuscation process generates expressions with predictable size increases.

Examining Figure 3a in comparison, the interquartile range spans approximately 18 to 34 nodes with a median value of 24. Whiskers extend from near 0 to 60 nodes, with outliers appearing around the 70-node mark. *GAMBA's* deobfuscation produces expressions larger than ground truth by a median of 24 nodes. This wide distribution indicates inconsistent deobfuscation results, where some expressions remain close to their original size while others maintain significant bloat.

As for *ProMBA*, Figure 3b showcases the interquartile range, spanning approximately from 27 to 40 nodes, indicating that the middle 50% of *ProMBA*-generated expressions fall within this size increase range. The vertical line within the box shows a median value of 33 nodes, representing the typical size increase compared to ground truth expressions. The whiskers extend from about 10 to 55 nodes, showing the general range of size deviations excluding outliers. Several outliers are visible on both ends of the distribution: three clustered near the 60-node mark, indicating some expressions with substantially larger size increases, and three on the left side (near 0 and 5 nodes), suggesting a few cases where *ProMBA* produced expressions much closer to the original ground truth size.

SaMBA demonstrates effectiveness at increasing expression complexity, generating obfuscated expressions with a median increase of 45 nodes compared to ground truth. This substantial size inflation creates expressions resistant to complete simplification, as evidenced by *GAMBA*'s deobfuscation results, which reduce but cannot eliminate this added complexity. The significant gap between deobfuscated expressions and ground truth (median 24 nodes) confirms *SaMBA*'s robust obfuscation capabilities. *ProMBA* occupies an intermediate position with a median size deviation of 33 nodes, situating it between *SaMBA*'s heavily obfuscated expressions and *GAMBA's* deobfuscated results. This placement indicates *ProMBA* generates moderately complex expressions that maintain greater complexity than *GAMBA's* output while avoiding the extreme bloat of *SaMBA*'s obfuscations.

The distribution characteristics provide additional insights into each technique's consistency. *SaMBA*'s narrower interquartile range compared to *GAMBA's* wider spread indicates the obfuscation process produces more predictable size increases than deobfuscation



(a) Size deviation of deobfuscated expressions from ground truth in *GAMBA*

(b) Size deviation of deobfuscated expressions from ground truth in *ProMBA*

(c) Size deviation of obfuscated expressions from ground truth

Figure 3: Comparison of expression size deviations: The top plots show *GAMBA* and *ProMBA*, while the bottom plot demonstrates that *SaMBA*'s obfuscation creates significantly larger expressions with a median increase of 45 nodes.

Method	Depth (Obf.)	Depth (GT)	Depth (Deobf.)	Recovery Rate
GAMBA	17.62	4.00	10.33	53.52%
ProMBA	17.62	4.00	14.24	24.82%

Table 4: Comparison of average DAG depths.

Method	Alt. (Obf.)	Alt. (GT)	Alt. (Deobf.)	Recovery Rate
GAMBA	86.74	0.33	4.49	95.29%
ProMBA	86.65	0.33	17.08	80.59%

Table 5: Comparison of average alternation complexity.

attempts. This consistency further validates *SaMBA*'s reliability in generating robust MBA obfuscations. Meanwhile, *ProMBA* exhibits outliers on both ends of its distribution, suggesting occasional expressions either approaching ground truth size or exhibiting significantly greater complexity, indicating somewhat inconsistent behavior compared to the other techniques. Collectively, these patterns demonstrate a clear hierarchy in expression complexity manipulation: *SaMBA* most aggressively increases size, *ProMBA* moderately increases size, and *GAMBA* reduces but cannot completely restore original expression complexity. The persistence of size deviation across all techniques, with expressions consistently larger than their ground truth equivalents, highlights the fundamental challenge in MBA expression manipulation and the effectiveness of advanced obfuscation techniques in resisting simplification by state-of-the-art deobfuscation tools.

DAG Depth. Examination of DAG depth metrics reveals complexity differences between original and processed expressions. DAG depth provides insight into expression complexity by measuring the longest path in the computational graph, directly corresponding to the sequential dependency chain required for evaluation.

The average DAG depth of obfuscated expressions reaches 17.62, representing more than four times the ground truth depth of 4. Following deobfuscation, *GAMBA* reduces the average depth to 10.33, while *ProMBA* achieves a less substantial reduction to 14.27. *GAMBA* achieves a recovery rate of 53.52% in expression depth, while *ProMBA* reaches 24.82%. Table 4 summarizes the results.

This metric indicates the minimum number of sequential operations needed, making it particularly relevant for parallel computation contexts and human comprehension difficulty. The persistent elevation of deobfuscated depths above ground truth values demonstrates that both tools, while partially effective, leave expressions with significantly deeper computational structures than their original forms, indicating high potency and resilience of our obfuscation.

Alternation Complexity. Beyond size and depth, alternation complexity provides a third criterion for evaluating MBA expression obfuscation effectiveness. The utilized ground truth expressions exhibit minimal alternation complexity (average 0.33), indicating that natural expressions rarely switch between operation domains. In contrast, SaMBA-obfuscated expressions demonstrate extreme alternation complexity (average 86.74), representing a 262-fold increase from ground truth.

Deobfuscation approaches show markedly different capabilities in reducing alternation complexity. *GAMBA* substantially reduces to an average of just 4.49 alternations, representing a 95.29% recovery rate. While still 13.6 times higher than ground truth, this represents significant progress toward the original expression structure. *ProMBA's* deobfuscation reduces complexity to a higher average of 17.19 alternations, achieving an 80.59% recovery rate, see Table 5.

This disparity in alternation reduction effectiveness highlights distinct optimization priorities between the analyses, with *GAMBA* showing greater capability in restoring simpler algebraic structures. Nevertheless, the alternation complexity of the deobfuscated expressions remains 13 times higher for *GAMBA* and 52 times higher for *ProMBA* compared to the ground truth.

When evaluating the complexity of the deobfuscated expressions, *GAMBA* consistently outperforms *ProMBA* across all metrics, suggesting superior deobfuscation capabilities. However, the metrics show different patterns of resistance to simplification. While

GAMBA achieves remarkable success in reducing alternation complexity (95.29% recovery rate), it makes substantially less progress in simplifying DAG depth (53.52%) and even less in reducing DAG size (41.8%). This shows that even when deobfuscation successfully reduces one dimension of complexity, others may remain resistant.

4.2.2 Syntia. Evaluation of Syntia's deobfuscation performance reveals a significant limitation in success rate, with only 10.5% of attempts yielding valid expressions. Among these successful cases, it demonstrated notable efficiency in expression simplification, achieving a maximum size reduction of 70.48%. This substantial disparity between success rate and optimization potential indicates that while Syntia possesses powerful simplification capabilities when applicable, its overall effectiveness is constrained by limited generalizability across diverse obfuscation patterns.

These results align with related work [25], where Syntia achieved similar success rates when deobfuscating larger datasets. Moreover, among the 105 successfully synthesized expressions, 39 are not semantically equivalent (37.15%), which further illustrates that Syntia cannot guarantee semantic equivalence as discussed in Section 4.1.1.

Takeaways

We showed with the batch evaluation that:

- *GAMBA* and *ProMBA* generated semantically correct expressions but failed to restore the original complexity.
- *Syntia* showed simplification potential but succeeded in only 10.5% of cases, with even fewer results being semantically equivalent.

Consequently, *SaMBA* produced robust MBAs that resisted full simplification by state-of-the-art deobfuscators.

4.3 Performance Evaluation

The utilized obfuscations on the benchmarked file had an average size of 56.78 and a depth of 17.22. On average, the additional execution time on an Intel i7-1165G7 was 0.028 seconds ($\sigma = 0.23$), corresponding to an overhead of 66.27% compared to the non-obfuscated sample. In absolute numbers, the difference was smaller for the 107 real-world examples, with an average of 0.006 seconds ($\sigma = 0.05$). In comparison, the seven random examples required more time, with an average of 0.381 seconds ($\sigma = 0.89$). This difference arises because `sha512sum` runs longer on larger files. For instance, computing the hash of a 1 GB random file required 3.63 seconds in the original version, whereas the obfuscated version took 6.04 seconds, resulting in a 66.37% overhead. The average overhead remained similar across the two groups, with 67.33% for the real-world examples and 66% for the randomly generated ones.

Although our obfuscation introduces some computational overhead, it remains moderate. State-of-the-art obfuscators such as *Loki* [26] report MBA implementations causing far higher slowdowns, with performance overhead factors of up to 167. The authors of *Loki* still considered it as acceptable in security-critical contexts. By comparison, our measured overhead factor is about 1.66 only. The cost can be reduced further by using fewer nodes, while still effectively hindering deobfuscation, as shown in Section 4.2. In practice, it is also common to obfuscate only sensitive functionality instead of entire programs.

Takeaways

By evaluating the performance of the obfuscated `sha512sum`, we showed that:

- *SaMBA* increased the execution time by 66%.
- Despite the extensive obfuscation and the computationally intensive algorithm, the code remained practically usable.

5 Limitation and Future Work

We focused on demonstrating that equality saturation increases MBA resilience against deobfuscation. We did not analyze the influence or resilience of individual rewrite rules because equality saturation applies all rules uniformly, and *SaMBA* treats these rules as exchangeable by design. A rule-level analysis could support the selection of rules. Thus, it is an interesting for future work.

We designed *SaMBA* for seamless integration into existing Tigress workflows. Thus, developers who already use Tigress can keep their workflows unchanged. Usability of defense mechanisms plays a crucial role, but a systematic evaluation of Tigress usability lies beyond the scope of this work, and we leave it to future research.

We evaluated *SaMBA* against state-of-the-art deobfuscation approaches. As it is common for defense mechanisms, the interaction between attackers and defenders forms an ongoing arms race [27]. We acknowledge that future deobfuscation techniques may overcome current limitations, resulting in more effective analyses.

6 Conclusion

We presented *SaMBA*, a methodology for systematically generating Mixed Boolean-Arithmetic (MBA) expressions using equality saturation and e-graphs. By employing a negative-size cost function and a comprehensive set of rewrite rules, *SaMBA* produces deeply nested and algebraically diverse expressions while preserving semantic correctness. This design ensures that obfuscation is both functionally sound and structurally resistant to simplification.

Our evaluation demonstrates that *SaMBA* defeats state-of-the-art deobfuscators, including stochastic synthesis (*Syntia*, *ProMBA*), algebraic simplifiers (*GAMBA*, *MBA-Blast*), pattern matchers (*SSPAM*), and bit-blasting approaches (*Arybo*). These either fail to recover semantics, generate syntactically complex results, or do not terminate on larger inputs. Even when they preserve semantic equivalence, the deobfuscated outputs remain substantially more complex than the ground truth, confirming *SaMBA* 's resilience.

By integrating our methodology with the Tigress obfuscator, we were able to demonstrate that *SaMBA*-generated MBAs can be deployed in real-world code. While obfuscation introduced moderate runtime overhead (66%), this remains far below slowdowns reported in comparable systems such as *Loki*, and it is well within practical bounds for security-critical applications.

Acknowledgments

The financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

[1] T. Blazytko, M. Contag, C. Aschermann, and T. Holz. 2017. Syntia: Synthesizing the semantics of obfuscated code. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*.

[2] C. Collberg. Documentation for Arithmetic Encodings in Tigress. https://tigress.wtf/encodeArithmetic.html.

[3] C. Collberg. Documentation for Encode Data in Tigress. https://tigress.wtf/encodeData.html.

[4] C. Collberg, C. Thomborson, and D. Low. 1997. A Taxonomy of Obfuscating Transformations. *Technical Report 148, Department of Computer Science, University of Auckland*.

[5] R. David, L. Coniglio, and M. Ceccato. 2020. QSynth - A Program Synthesis based approach for Binary Code Deobfuscation. In *Proceedings of the Workshop on Binary Analysis Research (BAR)*. doi:10.14722/bar.2020.23009.

[6] L. De Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[7] D. Detlefs, G. Nelson, and J. B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52, 3. doi:10.1145/1066100.1066102.

[8] N. Eyrolles. 2017. *Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools*. Theses. Université Paris Saclay (COmUE). https://theses.hal.science/tel-01623849.

[9] N. Eyrolles, L. Goubin, and M. Videau. 2016. Defeating MBA-based Obfuscation. In *Proceedings of the ACM Workshop on Software PROtection*. doi:10.1145/2995306.2995308.

[10] W. Feng, B. Liu, D. Xu, Q. Zheng, and Y. Xu. 2020. NeuReduce: Reducing Mixed Boolean-Arithmetic Expressions by Recurrent Neural Network. In *Findings of the Association for Computational Linguistics: EMNLP*. doi:10.18653/v1/2020.findings-emnlp.56.

[11] fvrmatteo. GitHub – NeuReduce Fork. https://github.com/fvrmatteo/NeuReduce.

[12] A. Guinet, N. Eyrolles, and M. Videau. 2016. Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions. In *Proceedings of GreHack*.

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

[14] R. Joshi, G. Nelson, and K. Randall. 2002. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/512529.512566.

[15] C. König, D. Schmidt, P. König, P. Felbauer, and S. Schrittwieser. GitHub – SaMBA Z3 Verification. https://github.com/CDL-AsTra/SaMBA/tree/main/z3_verification.

[16] J. Lee and W. Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. doi:10.1145/3576915.3623186.

[17] S. Lee, H. Jeon, and E.-S. Cho. 2024. Poster: E-Graphs and Equality Saturation for Term-Rewriting in MBA Deobfuscation: An Empirical Study. In *Proceedings of the 31st ACM SIGSAC Conference on Computer and Communications Security (CCS)*. doi:10.1145/3658644.3691382.

[18] C. Liem, Y. X. Gu, and H. Johnson. 2008. A Compiler-Based Infrastructure for Software-Protection. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. doi:10.1145/1375696.1375702.

[19] B. Liu, W. Feng, Q. Zheng, J. Li, and D. Xu. 2021. Software Obfuscation with Non-Linear Mixed Boolean-Arithmetic Expressions. In *Proceedings of the 23rd International Conference on Information and Communications Security (ICICS)*. doi:10.1007/978-3-030-86890-1_16.

[20] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu. 2021. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*.

[21] G. Menguy, S. Bardin, R. Bonichon, and C. d. S. Lima. 2021. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. doi:10.1145/3460120.3485250.

[22] G. Nelson and D. C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM (JACM)*, 27, 2. doi:10.1145/322186.322198.

[23] Oreans Technologies. Code virtualizer. https://oreans.com/help/cv/hm_codevirtualizer.htm.

[24] B. Reichenwallner and P. Meerwald-Stadler. 2022. Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions. In *Proceedings of the CheckMATE workshop, co-located with the ACM Conference on Computer and Communication Security*. doi:10.1145/3560831.3564256.

[25] B. Reichenwallner and P. Meerwald-Stadler. 2023. Simplification of General Mixed Boolean-Arithmetic Expressions: GAMBA. In *Proceedings of the 2nd Workshop on Robust Malware Analysis (WORMA), co-located with the 8th IEEE European Symposium on Security and Privacy*. doi:10.1109/EuroSPW59978.2023.00053.

[26] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*.

[27] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Computing Surveys*, 49, 1. doi:10.1145/2886012.

[28] R. E. Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM (JACM)*, 22, 2. doi:10.1145/321879.321884.

[29] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Logical Methods in Computer Science*, 7, 1. doi:10.2168/lmcs-7(1:10)2011.

[30] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proceedings of the VLDB Endowment*, 13, 12. doi:10.14778/3407790.3407799.

[31] H. S. Warren. 2013. *Hacker's delight*. Pearson Education.

[32] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages*, 5, POPL. doi:10.1145/3434304.

[33] D. Xu, B. Liu, W. Feng, J. Ming, Q. Zheng, J. Li, and Q. Yu. 2021. Boosting SMT Solver Performance on Mixed-Bitwise-Arithmetic Expressions. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/3453483.3454068.

[34] Y. Yang, P. Phothilimthana, Y. Wang, M. Willsey, S. Roy, and J. Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*.

[35] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Proceedings of the International Conference on Information Security Applications (WISA)*. doi:10.1007/978-3-540-77535-5_5.