Defeating MBA-based Obfuscation

Ninon Eyrolles, Louis Goubin, Marion Videau

Defeating MBA-based Obfuscation

Ninon Eyrolles
Quarkslab
Paris, France
neyrolles@quarkslab.com

Louis Goubin
UVSQ, Laboratoire de Mathématiques
Versailles, France
louis.goubin@uvsq.fr

Marion Videau
Quarkslab and LORIA
Paris and Nancy, France
mvideau@quarkslab.com

ABSTRACT

Mixed Boolean-Arithmetic expressions are presented as a strong protection in the context of data flow obfuscation. As there is very little literature on the analysis of such obfuscated expressions, two important subjects of interest are: to define what *simplifying* those expressions means, and how to design a simplification solution. We focus on evaluating the *resilience* of this technique, by giving theoretical elements to justify its efficiency and proposing a simplification algorithm using a *pattern matching* approach. The implementation of this solution is capable of simplifying the public examples of MBA-obfuscated expressions, demonstrating that at least a subset of MBA obfuscation lacks resilience against pattern matching analysis.

Keywords

Obfuscation, reverse engineering, mixed boolean-arithmetic expressions, expression simplification, pattern matching

1. INTRODUCTION

Both at the source and binary levels, various software obfuscation techniques exist to make a program difficult to understand while preserving its functionalities. Many ad hoc techniques have been proposed and discussed to achieve "scrambling up" the program structure, like the insertion of dead or irrelevant assembly code, procedure slicing that makes use of opaque constructs to jump back and forth above junk code, the use of function pointers, procedure merging, redundant and false return statements (see [6, 7] for details on standard obfuscation techniques).

During the last decades, the practical reverse engineering of binary programs was essentially performed by first manually identifying and understanding the way the observed code was obfuscated, and one common technique used is to perform "pattern matching" to identify similar obfuscations [19, 10].

The situation can be compared to what historically happened for cryptology, with new algorithms being designed on a regular basis, and new cryptanalytic techniques also being invented. As concerns symmetric cryptography, it is generally considered that this to-and-fro process between cryptography and cryptanalysis has converged towards a rather stable state, whereas asymmetric cryptography often has the additional advantage of providing security proofs relating the security of public key algorithms to the difficulty of well-known mathematical problems.

The present paper focuses on techniques based on the use of Mixed Boolean-Arithmetic (MBA) expressions. These are expressions that mix classical arithmetic expressions (addition, multiplication, ...) and boolean expressions (exclusive-or, and, or, ...). Interestingly, an MBA expression can be written in many equivalent ways. Each MBA corresponds to a family of (more and more complex) equivalent forms, and choosing among this family enables to complexify the MBA parts of any program. This method has already been described and used, but to the best of our knowledge, no techniques have been published to reverse the process of obfuscating a general expression, meaning going back to the initial form of the MBA.

Our work aims at filling this gap, and initiating a new line of research for what concerns the reverse engineering of such Mixed Boolean-Arithmetic expressions. As a first step, it is an important issue to define what simplifying an MBA expression means. A second challenge consists in estimating the complexity of such a simplification and the conditions that make it possible or not. The third goal is to provide new strategies to simplify MBA expressions.

Our contributions thus comprise both theoretical explanations about the difficulty of reverse engineering MBA expressions, and a practical methodology that is illustrated on concrete examples. Metrics about this notion of simplification are also introduced, and can be seen as a tool to guide the aforementioned strategies. As a result of our new methodology, we show that we are able to simplify already published examples that were thought to be very difficult to reverse. The code for our simplification tool, called SSPAM (for Symbolic Simplification with PAttern Matching), is available at `https://github.com/quarkslab/sspam`. Moreover, the beginning of an empirical characterization of weak MBA expressions can be obtained.

The Problem of Simplification.

Rewriting methods employing MBA expressions are used to obfuscate expressions or formulas, meaning that attacking this obfuscation is equivalent to *simplifying* the obfuscated formula in order to *understand* what it computes. Here,

SPRO'16, October 28 2016, Vienna, Austria

ISBN 978-1-4503-4576-7/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2995306.2995308

understand can have different meanings depending on the context of the attack, for example:

- identify distinctive constants or operations of a standard algorithm,
- associate a high level semantics to different parts of the formula,
- extract the formula or part of it and use it in another context, with different parameters,
- invert the function containing the formula.

On the other hand, the notion of *simplicity* is highly dependent on the context and on the representation of the expression: simpler can mean easier to compute, cheaper to store, and many other things. Since obfuscation intends to resist reverse-engineering, it is facing both automatic and human analysis, which means we are not looking for a perfect definition of what a simple expression is. Though, we may assess that simplifying an MBA-obfuscated expression is somehow close (if not equivalent) to finding the original expression of the non-obfuscated program in our case. Indeed, since the MBA obfuscation we consider is mostly conducted through rewriting, we aim at returning to a former state. We do not consider exceptional situations: for example, it is very unlikely that by simplifying, we produce a program simpler than the original. The idea of finding the original expression (or at least get close enough to it) is also present in the problematics of decompilation, with which we share this interest.

2. BACKGROUND

We introduce in this section the existing work regarding MBA expressions and expression simplification.

2.1 Polynomial MBA

In full generality, expressions mixing arithmetic and bitwise operators are already in use in broad contexts without being given a name. Any expression mixing arithmetic operators and bitwise ones, for example applying a boolean mask on an integer before an addition, fulfills the minimal requirements to be called an MBA. Moreover, any bitwise or arithmetic operator available in a processor might be used to construct an MBA expression.

However, if for a general characterization we do not exclude any existing arithmetic or bitwise operator, in this paper we will consider *polynomial* MBA expressions as defined by Zhou et al. in [31], since all the MBA expressions we have encountered up to now in the context of obfuscation are of this form. The definition also includes pure arithmetic or bitwise operators alone.

The list of available operators can vary between use cases. For example in [31], besides the *usual* operators $\{+, -, \times\}$ and $\{\wedge, \vee, \oplus, \neg\}$, signed and unsigned inequalities alongside signed and unsigned shifts are also considered. Other operators as shuffle or convolution are not taken into account, even if they are relevant in other contexts [27].

Definition 1. [31] An expression E of the form

$$E = \sum_{i \in I} a_i \left(\prod_{j \in J_i} e_{i,j}(x_1, \ldots, x_t) \right) \tag{1}$$

where the arithmetic sum and product are modulo 2^n, a_i are constants in $\mathbb{Z}/2^n\mathbb{Z}$, $e_{i,j}$ are bitwise expressions of variables $x_1, \ldots, x_t$ in $\{0,1\}^n$, $I \subset \mathbb{Z}$ and for all $i \in I, J_i \subset \mathbb{Z}$ are finite index sets, is a *polynomial Mixed Boolean-Arithmetic (MBA) expression*.

For example, the expression E written as

$$E = (x \oplus y) + 2 \times (x \wedge y) \tag{2}$$

is an MBA expression, which *simplifies* to $E = x + y$.

Since the MBA-obfuscated expressions we have studied so far rely on composing layers of MBA rewrites, the following statement exposed in [31] is essential to us : the composition of polynomial MBA expressions is still a polynomial MBA expression. This guarantees that we are only working with polynomial MBA expressions.

In the following, for conciseness when referring to an "MBA expression", it will stand for a polynomial MBA expression. Moreover, our study is limited to the most frequent operators: $\{+, -, \times\}$ and $\{\wedge, \vee, \oplus, \neg\}$. We deliberately chose not to address the subject of MBA inequalities, since it partially changes the issue to handle. Indeed, an MBA inequality is an assertion, and its value is either true of false. While this value can also be interpreted as a number, the situation is slightly different from an expression that can take a great range of values. Depending on the context, an attacker might just want to check if the inequality is satisfiable or not, instead of recovering a simpler form of the expression. This problem is related to the one addressed by Biondi et al. in [2], which we discuss further in Section 2.2.3.

The technique to obfuscate one or several operators using MBA expressions was first presented in [30, 31] and in various patents [12, 13, 11] with intersecting lists of authors. The process relies on two components:

- MBA rewrites: a chosen operator can be rewritten with an equivalent MBA expression, as can be seen in Expression (2).
- Insertions of identities: let us call e any part of the expression being obfuscated, then we can write e as $f(f^{-1}(e))$ with f any invertible function on $\mathbb{Z}/2^n\mathbb{Z}$. In the work of Zhou et al., f is an affine function.

Those two principles can be observed in the process of getting Expression (3), an example of an obfuscated MBA expression on two variables $x, y \in \{0,1\}^8$:

$$\begin{aligned} e_1 &= (x \oplus y) + 2 \times (x \wedge y) \\ e_2 &= e_1 \times 39 + 23 \\ E &= (e_2 \times 151 + 111). \end{aligned} \tag{3}$$

Then, the overall expression

$$(((x \oplus y) + 2 \times (x \wedge y)) \times 39 + 23) \times 151 + 111$$

stands for $x + y$.

This technique has proved to be quite popular in obfuscation, in real life settings ([22, 4, 12]).

2.2 Expression Simplification

Considering the general literature on expression simplification, we can retain two types of simplifications:

1. computing a unique representation for equivalent objects (canonical representation),

2. finding an equivalent, but simpler form ("simpler" being context-dependent).

The first type of simplification is the most studied in literature since it can prove equivalence of expressions and check for equality to zero. Nevertheless, it has already been noted [5, 3] that a canonical form may not always be considered as the simplest form, depending on the definition of simplicity (whether it is cheaper to store, easier to read, more efficient to compute, related to some high level semantics...). This implies that if both problems are related (as a solution for one could solve the other), they can be different because of the context. In our case, this context could be supplied by the attacker performing the reverse engineering.

2.2.1 Simplification of Arithmetic Expressions

We can easily illustrate these two types of simplification with examples from the computer algebra field concerning pure arithmetic expressions (namely polynomials), where there exist efforts in both canonical form and other simplifications related to the context. The canonical form of polynomials is the expanded form. Depending on polynomials, this form is not necessarily the most readable: for example, the expanded form on the right of Expression (4) can be easily considered as simpler than the original form on the left, whereas the factorized form on the left of Expression (5) is more readable than its expanded form.

$$(x-3)^2 - x^2 + 7x - 7 = x + 2 \tag{4}$$

$$(1+x)^{100} = 1 + 100x + \cdots + 100x^{99} + x^{100} \tag{5}$$

Those examples show the difficulty of defining in full generality the notion of simplicity, even when a canonical form is available. In computer algebra software such as Maple [20], several strategies are often offered to the user, who has to choose the one adapted to the objective. Nevertheless, there are standard simplification steps suitable to all strategies that can be constantly applied (e.g. $x \times 0 = 0$).

2.2.2 Simplification of Boolean Expressions

The same issues arise in the field of minimization of boolean functions and simplification of logic circuits [29]. While there exist several normal forms for boolean functions (CNF, DNF, ANF), those are not always relevant in the case of circuit simplification. Indeed, the goals of circuit simplification can be various: reducing the number of gates, the depth of the circuit, the fan-out of the gates, etc. We provide with Figure 1 an example of such a circuit simplification.

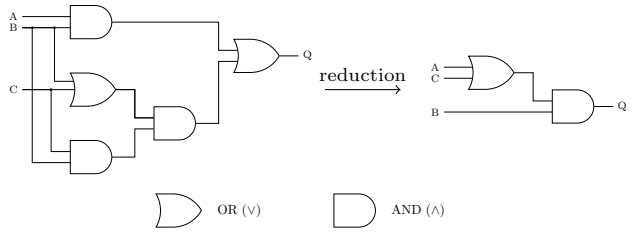

Figure 1: Example of circuit reduction from [17].

The formula corresponding to the circuit before reduction is $Q = (A \wedge B) \vee (B \wedge C \wedge (B \vee C))$. The reduced circuit coincides with a Conjunctive Normal Form (CNF), namely $(B \wedge (A \vee C))$, and as we can see, another normal form such as Disjunctive Normal Form (DNF), meaning $((A \wedge B) \vee (B \wedge C))$, would not have reduced the number of gates as much as the CNF. In some examples, CNF or DNF would probably not give the most simple circuit, depending on the considered characteristics. In fact, such variations and equivalences in circuit composition are indeed used to provide obfuscation at the circuit level [21].

2.2.3 Simplification of Mixed Expressions

To our knowledge, there exists only one article on simplifying MBA obfuscation, focusing on the obfuscation of a constant [2], for example to conceal a key or an opaque predicate. The authors use three techniques to recover the hidden constant: a SMT-based approach, an algebraic simplification technique and a drill-and-join synthesis method. While their focus is complementary to ours, their solutions do not scale well with our problem. For example, using a SMT solver in our case, would require to know the simplified expression and querying the solver would only validate the equivalence of both obfuscated and simplified expressions. Their algebraic simplification is too related to the form of the obfuscated constant, which is different from general obfuscated expressions, and the program synthesis approach seems too expensive for the general obfuscation case and at least requires further investigation as stated by the authors.

Regarding tools that could be used for MBA expressions simplification, existing tools often do not support both bitwise and arithmetic operators. Those implementing bitvector logic [16] allow at least the creation and manipulation of MBA expressions, but those tools are SMT solvers (e.g. Z3 [8] or Boolector [23]) and focus primarily on satisfiability, not necessarily on simplification. The function `simplify` of Z3 is rather sharing the same goal we have, but is very limited in the case of MBA (for example, it cannot simplify Expression (2)).

3. UTILITY OF MBA IN OBFUSCATION

We try to explore in this part different explanations for the resilience of MBA expressions as an obfuscation technique.

3.1 MBA in Cryptography vs in Obfuscation

Before being given the name of MBA in the context of obfuscation, such a mixing of bitwise and arithmetic operators was already used in the context of cryptography to design symmetric primitives with the stated goal of getting efficient, non-linear and complex interactions between operations. Building blocks such as ARX designs (e.g. [14]) and some generalizations can be found in major algorithms like hash functions (e.g. the SHA family), stream ciphers (e.g. Salsa family) or block ciphers (XTEA).

The notion of T-functions [15] appears both in the context of cryptography and obfuscation, as integer arithmetic operators, such as $(+, -, \times)$ are triangular T-functions and provide efficient non-linear invertible functions.

However, there is a key difference between what is looked for in cryptography and in obfuscation regarding MBA. In cryptography the MBA expression is the direct result of the algorithm description and the resulting cryptosystem has to verify a set of properties (e.g. non-linearity, high algebraic degree) from a black box point of view. The complex form of writing is directly related to some kind of hopefully in-

trinsic computational complexity for the resulting function: one wants the inverse computation without knowing the key to be intractable. In obfuscation, an MBA is the result of rewrite iterations from a simpler expression which can have very simple black box characteristics. There is no direct relation between the complex form of writing and any intrinsic computational complexity of the resulting function: on the contrary, when obfuscating simple functions, one knows that the complex writing is related to a simpler computational function. Nevertheless, getting the result of the computation for the obfuscated expression requires indeed to get through all the operators in the considered expression which implies somehow a computational complexity.

Therefore cryptography can provide us with an example study of what means incompatibility between operators and how it can prevent an easy study of MBA expressions in a unified domain. Indeed, we work on n-bit words considered at the same time as elements of different mathematical structures. For example, standard arithmetic operations are considered in $(\mathbb{Z}/2^n\mathbb{Z}, +, \times)$ while bitwise operations belong to $(\{0,1\}^n, \wedge, \vee, \neg)$ or $(\{0,1\}^n, \wedge, \oplus)$.

3.2 An Example Study of Incompatibility between Operators

IDEA [18] is a well known block cipher of the 90s, famous for its combined use of integer arithmetic and bitwise operators. One of the major characteristics of IDEA at the time of its proposal was its lack of S-boxes. Instead it relied on a construction which uses as key components:

- the multiplication $\odot$ in $\mathbb{Z}^*_{2^{16}+1}$,
- the addition $\boxplus$ in $\mathbb{Z}_{2^{16}}$,
- the bitwise XOR $\oplus$ in $\mathrm{GF}(2)^{16}$,

carefully interleaved so as to prevent any easy manipulation of the resulting expressions.

Even though the multiplication in $\mathbb{Z}^*_{2^{16}+1}$ is not part of the operators we consider in an MBA expression, IDEA provides us with a detailed example of incompatibility between operators which hopefully helps one understand the usefulness of MBA for obfuscation. The incompatibility study of the operators in IDEA was at the basis of the argument on the *confusion* property a block cipher must fulfill.

There are four main reasons why the three operations are incompatible [18]:

- No pair of the three operations satisfies a distributive law.
- No pair of the three operations satisfies a generalized associative law.
- When considering the quasi-groups, $(\mathbb{Z}^*_{2^{16}+1}, \odot)$ and $(\mathrm{GF}(2)^{16}, \oplus)$ are not isotopic, neither are $(\mathbb{Z}_{2^{16}}, \boxplus)$ and $(\mathrm{GF}(2)^{16}, \oplus)$. The isotopism between $(\mathbb{Z}^*_{2^{16}+1}, \odot)$ and $(\mathbb{Z}_{2^{16}}, \boxplus)$ is essentially the discrete logarithm, which is not a simple and straightforward bijection.
- It is possible to analyze $\boxplus$ and $\odot$ as acting on the same set. However it means either analyzing a non polynomial function on $\mathbb{Z}_{2^{16}}$ to represent $\odot$ or analyzing a high degree polynomial on $\mathbb{Z}^*_{2^{16}+1}$ to represent $\boxplus$.

The idea behind the notion of *confusion* is to make any description of the relation between the ciphertext, the plaintext and the key so involved and complex that it is useless for the attacker. It is a clear link with obfuscation concerns, although the starting point for each context is different: hopefully intrinsic computational complexity in cryptography and on the other hand, rewrite complexity in obfuscation preventing simplification.

3.3 Analysis Difficulties

As explained in previous sections, bitwise and arithmetic operators do not naturally interact very well, as there are no general rules (e.g. distributivity, associativity...) to cope with this mixing of operators. Though there are some cases where rules analogous to distribution can be used (e.g. Expression (6)), the impossibility of generalizing such rules (see Expression (7)) supplies additional diversity during the obfuscation process.

$$\forall x, y \in \mathbb{Z}/2^n\mathbb{Z}: \quad 2 \times (x \wedge y) = (2x \wedge 2y) \tag{6}$$

$$\exists x, y \in \mathbb{Z}/2^n\mathbb{Z}: \quad 3 \times (x \wedge y) \neq (3x \wedge 3y) \tag{7}$$

Moreover, as we presented in Section 2.2.3, there are very few public tools offering the possibility of MBA manipulation, and those tools being SMT-Solvers, they do not aim at simplifying an expression. While the function `simplify` of Z3 might give results going in our direction, as it can transform for example $(x \vee y) + (x \vee y)$ in $2 \times (x \vee y)$, it shows little efficiency in the simplification of MBA expressions, as for example on Expression (2).

Finally, the simplification of MBA expressions arises in the process of reverse engineering an obfuscated program. This means that, after the obfuscation of the program in itself, the analyst faces two more phases affecting the expression to simplify:

- Compilation: it is very likely that the MBA obfuscation occurred before or during the compilation of the program, and more importantly before the optimization passes of the compiler. Optimization passes are very numerous and may often change the MBA expressions in ways that are difficult to anticipate. This means that knowing only the obfuscation steps might not be enough to understand completely how the resulting expression was obtained, if the analyst needs such comprehension.
- Extraction of expressions: as the analyst works on a binary program, some tool is needed to obtain expressions from the assembly language. The most common way to do this is to use *symbolic execution*, provided by reversing frameworks such as Miasm [9] or Triton [26]. But this tool also influences the resulting expression by its choice of semantics, representation and eventually its own simplification passes.

Both of those reversing steps add complexity to the simplification of obfuscated MBA. In this paper, we use either a "clean" mathematical context where we do not consider compilation and extraction, or a reversing context on an expression extracted from a compiled program.

4. A NEW PROPOSAL FOR MBA SIMPLIFICATION

There are several leads for MBA expressions simplification. One technique exploits the bit-vector representation of MBA expressions, called *bit-blasting*. It consists in computing the boolean expression corresponding to each bit of the obfuscated formula. This approach has the advantage of moving the simplification problem from mixed algebra into boolean algebra, with known canonical forms. But it presents two major drawbacks: the cost of the reduction increases with the number of bits, and to our knowledge, there is no easy algorithm to identify the word-level formula corresponding to the n boolean expressions (especially for arithmetic operators).

We chose to stay on the word-level while designing our simplification algorithm to avoid such problems. In this section, we propose our own metrics to help define what a simplification algorithm should reduce when simplifying an obfuscated expression, then we detail our simplification algorithm as well as its implementation.

4.1 Simplicity Metrics

To represent MBA expressions, we use the *term graph* representation as defined in [24]. We use the compact representation for more clarity, which can informally be defined as an acyclic graph G where:

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

We present an example of the term graph of an MBA expression in Figure 2.

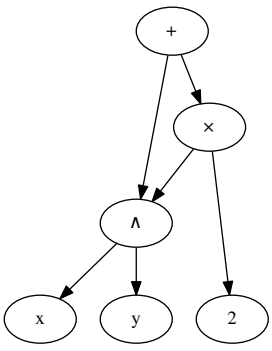

Figure 2: Term graph for the expression $2 \times (x \wedge y) + (x \wedge y)$.

We use two metrics based on this graph representation, arguing that decreasing these metrics improves the simplicity of the expression in a general way, both for human understanding and automatic analysis.

4.1.1 Number of Nodes

Reducing the number of nodes of the graph contributes to reducing the expression size, meaning it will be easier to apprehend and manipulate. Reducing the number of variables can also be interesting for any brute force approach, by reducing the size of the input set.

4.1.2 MBA Alternance

This metric is intended to help quantify the MBA aspect of an expression. For example, a purely arithmetic or a purely boolean expression has a null MBA alternance. Likewise, a computer algebra software applying only arithmetic simplifications (like expansion) on an MBA-obfuscated formula should not greatly decrease the MBA alternance metric of a robust obfuscation.

To define the MBA alternance of an expression, we first need to define the *type* of an operator *op*. The type is arithmetic if $op \in \{+, -, \times\}$, and the type is boolean (or bitwise) if $op \in \{\wedge, \vee, \oplus, \neg\}$. The MBA alternance is simply the number of edges linking two nodes that represent operators of different types (variables and constant nodes do not have a type, and thus do not affect this metric).

Definition 2. For a graph $G = (V, E)$ with V the set of vertices and E the set of edges, the *MBA alternance* $alt_{MBA}(G)$ is:

$$alt_{MBA}(G) = |\{(v_1, v_2) \text{ such that } type(v_1) \neq type(v_2)\}|,$$

where $(v_1, v_2) \in E$ represents the edge linking the two vertices $v_1, v_2 \in V$.

For example, Expression (3) has 15 nodes and an MBA alternance of 2.

One may note that some bitwise operators (e.g. bitwise not $\neg$, left shift $\ll$) can be rewritten as arithmetic expressions quite easily: for example, $\neg x = -x - 1$ and $x \ll n = x \times 2^n$, while there exists no simple equivalence for other bitwise operators ($\oplus, \wedge \ldots$). One may use such rewritings in the simplification process.

At the moment, we use those two metrics mainly to evaluate the relevance of our simplification algorithm (see Section 5), but we would like to use them more in the algorithm itself to guide the simplification (e.g. choose between several applicable rewriting rules).

4.2 Algorithm

As we discussed in Section 2.2, there exist both theoretical ground and tools to manipulate and simplify arithmetic expressions (e.g. polynomial expansion, factorization) or bitwise expressions (e.g. CND, DNF). While there is no such thing for MBA expressions yet, it is still possible to use existing simplification techniques on parts of the MBA that may contain only one type of operator. To create the missing link between alternating sub-expressions, one may use *term rewriting*. From [30, 13], we gather that the obfuscation technique mainly consists of rewriting operators with known equivalent MBA expressions. Knowing that, it comes naturally to use the same process of rewriting as a deobfuscation technique, by inverting rules used for obfuscation.

Term rewriting [1] is a process that needs rewrite rules, rules we can infer by orienting equalities (in our case, the orientation shows whether we consider obfuscation or deobfuscation). We represent that process with a binary relation $\rightarrow$. For example, we stated with Expression (2) that

$x + y = (x \oplus y) + 2 \times (x \wedge y)$; from that equality, we can deduce two rewrite rules:

$$x + y \rightarrow (x \oplus y) + 2 \times (x \wedge y) \quad (8)$$
$$(x \oplus y) + 2 \times (x \wedge y) \rightarrow x + y \quad (9)$$

The relation (8) would very likely be used for obfuscating purposes, since it increases both the number of nodes and the MBA alternance of the expression. Knowing this, it is conceivable to use rewrite rule (9) to invert the obfuscation process. All the rewrite rules we have encountered are invertible as they derive from an equality expression verified for all values of the input variables. If we restrict ourselves to the case where we want to find strictly equivalent expressions, valid for all values of the variables on some input range (which can be a fixed subset of values as long as it is known), we can safely consider that the rules used in the obfuscation process are all invertible.

If we keep analyzing the obfuscation technique, we can see in [30, 13] that it contains this type of rewriting as well as insertion of affine functions whose composition is equivalent to identity—to see an example of this obfuscation process, one can refer to Expression (3). In order to deobfuscate such expressions, one can first use a computer algebra software to compute the composition of affine functions and then use a list of rewrite rules to transform the resulting MBA expression into another simpler and equivalent expression. Note that regarding this obfuscation technique, we do not need a bitwise simplification step, since rewriting and arithmetic expansions are enough.

We provide with Figure 3 and Figure 4 a step-by-step example of the deobfuscation process on Expression (3).

Step one: MBA Rewriting.

The first step consists in using known rewrite rules to transform an MBA expression into a simpler expression (by reducing the number of nodes and the MBA alternance). In our example, the rewriting reduces the number of node from 17 to 11. This step also decreases the MBA alternance, from 2 to 0 here.

All the rewriting rules we encountered in the literature (e.g. [28, 31]) can be applied regardless of the number of bits of the expression.

$$\begin{aligned} e_1 &= (x \oplus y) + 2 \times (x \wedge y) \\ &= x + y \\ E &= (e_1 \times 39 + 23) \times 151 + 111 \\ &= ((x + y) \times 39 + 23) \times 151 + 111 \end{aligned}$$

Figure 3: Rewriting MBA expressions.

Step two: Arithmetic Simplification.

The second step is to compute the composition of affine functions. This is in theory possible with any computer algebra software, given that it supports declaration of MBA expressions (more on this in Section 4.3). The composition of affine functions being equivalent to identity, this step drastically decreases the number of nodes: in this example it gets from 11 nodes to 3 nodes.

$$\begin{aligned} E &= ((x + y) \times 39 + 23) \times 151 + 111 \\ &= (x + y) \times 1 + 0 \\ &= (x + y) \quad \text{with } x, y \text{ of 8 bits} \end{aligned}$$

Figure 4: Computing composition of affine functions.

Since the obfuscation phases can be applied iteratively, those simplification steps can also be applied repeatedly until a fixed point is attained. It is relatively safe to assume that such a fixpoint will be reached if all the rewriting rules reduce the size of the expression (the arithmetic simplification step is sure to finish since it expands the expression).

The main drawback of this approach is that it is highly dependent on the chosen set of rewrite rules. Indeed, if only one obfuscation rule is unknown, the simplification algorithm is not able to reduce the expression as much as it would with knowledge of that rule. As the obfuscation process usually has a constraint of not deteriorating greatly the performances of the program being obfuscated, we can assume that the set of MBA rewrites will be of "reasonable" size. Furthermore, for a given expression size there is a limited amount of MBA rewrites, and eventually all these rules will be recovered by analysts.

4.3 Implementation

In this part, we detail the implementation aspects of SSPAM, mostly on the term rewriting (often called *pattern matching*): we expose in this section our choices of design. As we mentioned previously, most computer algebra software do not even support symbolic MBA expressions. Regarding reverse engineering software, there exist frameworks with expression manipulation that could handle MBA expressions (e.g. Miasm [9]), but the cost to enter such a complex framework was too high for us, since we mostly wanted to assess the feasibility of this kind of pattern matching.

To implement this proof of concept, we have made the choice to work on the *Abstract Syntax Tree* (AST) of the expression, using the Python module `ast`. This way, we can support any kind of expressions and rely on a stable and simple module. Even if this implies less features than in a big framework, it also implies we control all the process of pattern matching. The simplification algorithm consists of two main components, related to the simplification steps proposed in Section 4.2: pattern matching and arithmetic simplification.

4.3.1 Pattern Matching

This part concerns the implementation of the term rewriting presented in Section 4.2; by *pattern matching*, we include pattern detection and rewriting. As we wanted our pattern matcher to be as general as achievable while controlling as much as possible its behavior, we implemented the whole process.

The pattern matching step is applied before the arithmetic simplification for two reasons: firstly, affine functions help delimitate the expression potentially subjected to a rewrite, since in the obfuscation process they are often applied after the rewrite part. Secondly, the rewriting step might bring expressions obfuscated with arithmetic operators (for example, a classical way to greatly increase the size of an

expression is to repeat the same term, e.g. $x \to x + x - x$). The arithmetic simplification step helps simplify such expressions.

We also implemented what we call *flexible* matching, which allows to match a pattern when the expressions appear different but are equivalent. For example, let us consider the rewrite rules (10) provided in [30] (slightly modified for readability purposes). The pattern matching will search for a formula matching the left hand side (LHS) term of the rewrite rules, with x and y any kind of expression.

$$\overbrace{(x \oplus (\neg y)) + 2 \times (x \vee y)}^{\text{pattern}} \to x + y - 1 \tag{10}$$

Considering the two examples (on 8 bits) of (11) and (12), we can see that matching the LHS of rule (10) with example (11) is straightforward, by substituting x with $(a + 9)$ and y with $(b \oplus 23)$. On the other hand, it is not trivial to detect the same pattern in example (12) (with the same substitution for x and y). Indeed, we have $\neg(b \oplus 23) = (b \oplus \neg 23) = (b \oplus 232)$ on 8 bits, which means that both Expressions (11) and (12) are computationally equivalent and can both be rewritten according to rule (10). Nevertheless, one matching is trivial and the other requires to prove the equivalence of $\neg(b \oplus 23)$ and $(b \oplus 232)$.

$$((a + 9) \oplus (\neg(b \oplus 23))) + 2 \times ((a + 9) \vee (b \oplus 23)) \tag{11}$$

$$((a + 9) \oplus (b \oplus 232)) + 2 \times ((a + 9) \vee (b \oplus 23)) \tag{12}$$

To deal with this type of situation, we use a SMT solver (Z3 for example) to prove that both instances are equivalent and match the pattern. This problem occurs especially because of the optimization phase (as mentioned in Section 3.3), mainly due to the optimization pass called *constant folding*. Consulting the SMT solver at each step of the pattern matching would be very costly, and thus the pattern matcher only does it when encountering certain patterns that we know to be subject to constant folding (e.g. $(\neg x)$, $2 \times x \dots$).

The patterns currently present in SSPAM are those of public knowledge [30, 31, 28] and some found from our own analysis. The tool also offers the possibility to the user to add his or her own patterns.

4.3.2 Arithmetic Simplification

The arithmetic simplification component is used to compute the composition of affine functions and apply classical arithmetic simplifications on MBA expressions (as explained in the previous paragraph). This step represents what computer algebra software can handle at the arithmetic level, therefore we decided to use an existing solution for this component, instead of implementing our own module. We use the Python module `sympy` that offers symbolic computations using the Python language. It does not support MBA expressions, but by defining every bitwise operator as an unknown function, we were able to use the arithmetic simplification engine of `sympy`.

5. EVALUATION

In this section, we try to estimate the efficiency of our simplification approach.

5.1 Methodology

We first tested SSPAM on public MBA-obfuscated examples [30, 22]. Then, to try to categorize the strengths and weaknesses of the obfuscation, we generated our own samples (thanks to a Python obfuscator we implemented following the process of [31, 12]), by considering either rewrites only or full process of rewrites and identities insertion.

Also, as explained in Section 3.3, the expression usually analyzed comes from a context of reverse engineering, meaning compilation and symbolic execution have probably modified it after obfuscation. In order to assess the resilience of the MBA obfuscation technique both in itself, and in a reverse engineering context, we conducted the evaluation of our simplification tool on two types of inputs:

- Mathematical context: expressions in Python language directly obfuscated with our Python obfuscator.
- Reverse engineering context: expressions in C language generated from the Python obfuscator, compiled with GCC [25] with the optimization option `-O3`. To extract an exploitable expression from the compiled program, we used the framework Miasm.

Here, we consider that the output of the simplification tool is fully simplified when it returns the original expression (which we know to be only one operator for the public examples, and is of course known for our generated expressions).

In the following experiments, we only try to determine the resilience of the MBA obfuscation as defined in [31], and used alone without other control flow or data flow obfuscation techniques. If several layers of obfuscation were to be used, the difficulty of simplifying the expression would greatly increase, and the analyst would very probably need to deobfuscate each layer separately.

5.2 Results

5.2.1 Simplifying the State of the Art

We applied SSPAM on the few public obfuscated expressions available in the literature:

- All examples of obfuscated operators of Zhou et al.'s work [30] were fully simplified by our tool. A comparison of obfuscated inputs and simplified outputs can be found in Figure 5. Simplifying those examples takes between one and three seconds with our tool.
- A larger example of an MBA-obfuscated expression found in a real-life obfuscated DRM was given in [22] and reproduced here in Figure 6. We were able to retrieve the original expression $(x \oplus 92)$ with our tool in about 12 seconds.

5.2.2 Obfuscation with Rewriting only

To further validate the strategy of simplification with rewriting, we obfuscated short expressions ($(x+y)$ and $(x \oplus y)$) by choosing a random node and rewriting it with an equivalent MBA expression (there were four rules possible, one for each operator $+, \oplus, \wedge, \vee$), going up to 100 rewriting steps.

In the mathematical context, the pattern matching process (without the flexible part) was enough to simplify these expressions. Obfuscating expressions containing constants

$$
\begin{aligned}
t1 = {} & (4211719010 \oplus 2937410391 * x) + 2 * (2937410391 * x \vee 83248285) + 4064867995 \\
t2 = {} & (2937410391 * x \vee 3393925841) - ((2937410391 * x) \wedge 901041454) + 638264265 * y \\
z = {} & 519915623 * t1 - ((3383387769 * t2 + 129219187) \oplus 2756971371) \\
& - 2 * ((911579527 * t2 + 4165748108) \vee 2756971371) + 4137204492
\end{aligned}
$$

(a) Obfuscated expressions [30].

$$
\begin{aligned}
t1 &= ((2937410391 * x) + 4148116279) \\
t2 &= ((638264265 * y) + 3393925841) \\
z &= (x + y)
\end{aligned}
$$

(b) Outputs of SSPAM.

Figure 5: Simplification of some state of the art examples.

$$
\begin{aligned}
a &= 229x + 247 \\
b &= 237a + 214 + ((38a + 85) \wedge 254) \\
c &= (b + ((-2b + 255) \wedge 254)) \times 3 + 77 \\
d &= ((86c + 36) \wedge 70) \times 75 + 231c + 118 \\
e &= ((58d + 175) \wedge 244) + 99d + 46 \\
f &= (e \wedge 148) \\
g &= (f - (e \wedge 255) + f) \times 103 + 13 \\
result &= (237 \times (45g + (174g \vee 34) \times 229 + 194 - 247) \wedge 255)
\end{aligned}
$$

Figure 6: MBA-based obfuscation of $(x \oplus 92)$.

(either in the original expression or introduced by the rewriting rules) did not add any further difficulty for the pattern matching process. In the reverse process however, while the pattern matching is sufficient for expressions containing only variables, arithmetic simplification and flexible matching must be activated as soon as constants are involved, a plausible explanation being that constants enable more optimization possibilities during the compilation process.

5.2.3 Obfuscation with Rewriting and Identities

We have generated obfuscated expressions with the full process: MBA rewrites and insertion of affine functions whose composition is equivalent to identity. What we observed is that even in the mathematical context, insertion of affine functions implies the need of arithmetic simplification, which has a lot of side effects on the expression being simplified and thus requires the use of flexible matching.

The obfuscation process as we implemented it is described in Algorithm 1.

Using this function for MBA obfuscation, we chose four expressions to obfuscate, $(x+y)$, $(x \oplus y)$, $(x \wedge 78)$ and $(x \vee 12))$ on 8 bits, four rewriting rules from [28] (given in Figure 7), one for each operator $+, \oplus, \wedge, \vee$, and for each degree from 1 to 10, generated 50 obfuscated expressions to be used as input for SSPAM. We then computed the number of fully simplified expressions and the average number of node reduction (from obfuscated expression to simplified expressions) as well as the average MBA alternance reduction.

The average number of node reduction can be seen in Figure 8. The 100% ratio in degree one means that those obfuscated expressions were fully simplified (in the other degrees, fully simplified expressions are not accounted for in the average computation to get a better idea of the behavior of not completely simplified expressions). The results for average MBA alternance reduction were very similar, we thus chose not to add them into this paper.

The number of fully simplified expressions (on 50 tests) depending on the degree of obfuscation is detailed in Figure 9. We did not include the results for degree 1, since for most of the input expressions, all 50 obfuscated expressions were fully simplified.

Algorithm 1 MBA-Obfuscation Algorithm.

Require: expression e, degree of obfuscation d, number of bits n, a list of rewrites rules R
Ensure: obfuscated expression e'

```
function MBA-Obf(e, d, n, R)
    loop d times
        Choose a random operator of e
        Choose a random rule r ∈ R for this operator
        Rewrite this operator with r
        Choose two random coefficients a, b on n bits for the affine
        while a non-invertible modulo 2^n do
            Choose randomly another a
        end while
        Compute a^{-1} and -ba^{-1} coefficients of the inverse affine
        Insert affines composition around the rewritten operator
    end loop
end function
```

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

Figure 7: Rewriting rules used to obfuscate our sample expressions.

From the results of Figures 8 and 9, one can see that if the degree seems to determine largely the number of fully simplified expressions, it does not influence greatly the average number of node reduction, being around 50% for all four input expressions. The average MBA alternance reduction follows the same pattern, which can be linked to the fact that all the rewriting rules we used reduce both of those metrics. One can also notice the particular case of the obfuscation of $(x \wedge 78)$, that could not be fully simplified in degree one. This is due to some case of associativity in the flexible pattern matching that is not yet implemented in our tool, and shows that even with simple cases, it can be hard to consider every form of the pattern.

When analyzing the obfuscated expressions not fully simplified by SSPAM, we determined that a strategy using factorized form (instead of the expanded one) during the arithmetic simplification should provide an expression manage-

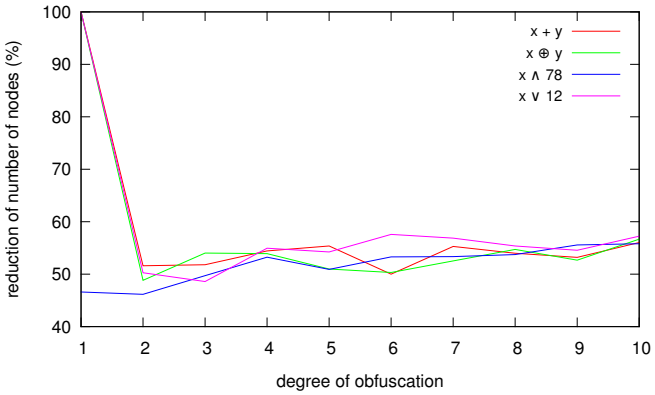

Figure 8: Average Number of Nodes Reduction.

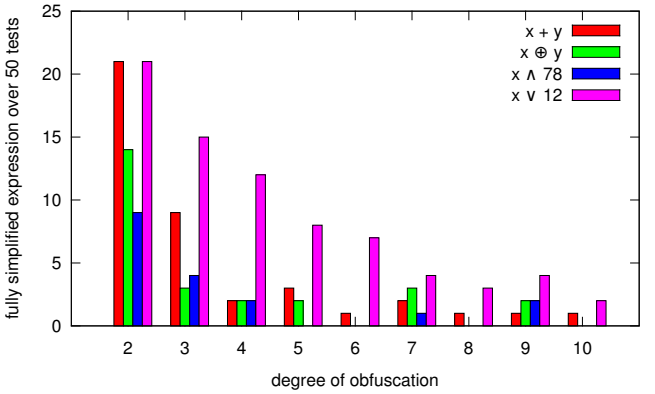

Figure 9: Number of fully simplified expressions.

able by the pattern matcher. The implementation of this strategy should help increase the number of fully simplified expressions for high degree.

5.3 Discussion

The results regarding obfuscation with MBA rewrites only, presented in Section 5.2.2, allow us to conclude that an obfuscation technique using only MBA rewriting would be quite weak, as a simple pattern matching process appears to be very efficient to fully simplify such obfuscated expressions (considering that the rewriting rules are all invertible).

The insertion of compositions of affine functions contributes to the resilience of the obfuscation by forcing the use of an arithmetic simplifier in addition to the pattern matching process, which leads to side effects that may be difficult to anticipate. The flexible pattern matching proves all its efficiency when the arithmetic simplification is used, by dealing with most of these side effects.

Regarding the MBA-obfuscated expressions that we generated and gave as input to SSPAM, the average number of nodes reduction of 50% and the number of fully simplified expressions for small degrees shows that our approach can prove itself efficient to deobfuscate such expressions. The implementation of more strategy (e.g. factorization) would help to improve those results.

It is clear that this simplification algorithm relies a lot on the found patterns, but we can assume that for practical reasons, the list of obfuscation rewrite rules is not likely to be huge.

Furthermore, our simplification tool is designed to help the analyst in his or her simplification process by automating steps, and offers the possibility to add custom rules detected by the analyst.

As we were able to fully simplify all examples publicly available, we can conclude that our simplification algorithm proves to be efficient in breaking existing MBA-based obfuscation, and that the MBA obfuscation technique, at least as it is basically presented in practice, does not offer a great resilience. Several ideas can be explored to increase the difficulty of simplification.

- The use of obfuscation rewrite rules where the right-hand side generates several simplification rewrite rules when put on the left-hand side: consider for example $2\times(7\wedge x) \rightarrow 14\wedge 2\times x$. The simplification rules could be either $14\wedge 2\times x \rightarrow 2\times(7\wedge x)$ or $14\wedge 2\times x \rightarrow 2\times(135\wedge x)$. More generally, rewrite rules relying on the very precise structure of constant values may be more numerous and difficult to simplify by a general strategy as it can be the case with rules implying variables only.

- The use of different types of identity expressions: invertible polynomials as defined in [31] (instead of affine functions) would make it more difficult to use arithmetic simplification to compute the compositions.

6. FUTURE WORK

Several issues remain to be addressed to refine our results.

6.1 Definition of Simplicity

There is no good general definition for MBA simplicity, only a set of metrics. The complexity of this subject lies in the fact that simplicity is related to the context where the expression is considered. For example, a factorized polynomial is more convenient to search for roots, while an expanded form might be shorter, or allow for the recognition of patterns. But on the other hand, factorized form may enable to detect patterns for further factorization, while the expanded form is the most appropriate to check for equality to zero. The notion of readability is close to the one of pattern recognition. Indeed, Expression (13), while less compact than Expression (14), might be considered as simpler because it can be described in a more concise way, i.e. $\sum_{i=0}^{10}(i+1)x^i$.

$$1 + 2x + 3x^2 + 4x^3 + \cdots + 11x^{10} \quad (13)$$

$$1 + 3x + 4x^2 + x^3 - 9x^4 + 5x^5 + x^6 + 2x^7 \quad (14)$$

This example shows how numerous are the aspects that influence the definition of the simplicity of an expression.

We do not aim at giving an absolute definition of simplicity. A characterization depending on the context in which the expression is manipulated could shed some light on our problematics. In the deobfuscation use case, the context might be the purpose of the attack: for example, either identifying a distinctive feature of a standard algorithm or extracting a formula.

6.2 Properties of Rewrite Rules

Considering the theory of term rewriting, there are two standard properties interesting to prove for a set of rewrite rules: *termination* and *confluence*.

Termination guarantees that after finitely many rules applications, we always reach an expression to which no more rules apply. Regarding the simplification with MBA rewrites, we can suppose that each rule decreases the number of nodes of the expression (we failed to find an obfuscation rule that would decrease the size of the expression, implying a simplification rule that would increase it). This is enough to guarantee the termination of the single rewriting part of the algorithm. But it would be interesting to study how this interfaces with the arithmetic simplification part of the algorithm, which is also guaranteed to terminate because it computes an expanded form. For now, we are rather confident that the termination of each component implies the termination of the whole simplification algorithm, but they may be some corner cases refuting this assertion.

Confluence ensures that if there are different rules to apply to a term t leading to two different terms t_1 and t_2, we can always find a common term s that can be reached from both t_1 and t_2 by application of rewrite rules. We did not look into confluence too much, but it would probably be much harder to prove for a class of rewrite rules instead of a specific set.

A set of rules having both termination and confluence is sure to produce a canonical form, which would greatly help in the simplification process. Managing to prove those properties even for a subset of rewrite rules would probably allow us to validate our empirical results.

6.3 Improving SSPAM

Our simplification tool is constantly improved thanks to the different obfuscated examples we encounter or generate. The two main features that we plan to implement are:

- Strategy using factorization instead of expansion: the arithmetic simplification step could factor the expression to see if any pattern can be matched in this form.
- Bitwise simplification: in the model of the arithmetic simplificator, implement a bitwise/boolean simplificator that would conduct basic simplifications such as $x \vee 0 = x, x \wedge x = x$, as well as compute constant bitwise parts (e.g. $(x \wedge 115 \wedge 78 = x \wedge 66$ on 8 bits). Future obfuscation might combine boolean obfuscation with MBA rewrites, making this bitwise simplificator useful.

7. CONCLUSION

The obfuscation technique using MBA expressions was thought to be resilient, mostly because of the absence of theoretical ground or tool to manipulate and simplify it, as boolean and arithmetic operators do not interact well. We have shown that the definition of simplification itself is not trivial and highly depends on the context of the analysis. In this paper, we tried to assess the resilience of the technique from a theoretical point of view in Section 3 and we presented a simplification solution to reduce obfuscated expressions (ideally retrieve the original expressions) in Section 4. As a result, we were able to fully simplify all examples provided in the literature [30] and encountered in real life settings, while also trying to categorize the obfuscation steps adding resilience by generating our own obfuscated expressions with different techniques (see Section 5.2). Our conclusion is that practical MBA-based obfuscation that are available today as presented in the work of Zhou et al. does not offer a great resilience. However there is still further work to be done to determine which MBA obfuscations are practical and resilient, and construct more theoretical and practical tools to analyze them.

8. REFERENCES

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Aug. 1999.

[2] F. Biondi, S. Josse, A. Legay, and T. Sirvent. Effectiveness of Synthesis in Concolic Deobfuscation. Preprint, Dec. 2015.

[3] B. Buchberger and R. Loos. Algebraic Simplification. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra*, volume 4 of *Computing Supplementa*, pages 11–43. Springer, 1982.

[4] V. Bukasof and D. Schelkunov. Deobfuscation and beyond. ZeroNights conference, 2014. http://www.slideshare.net/ReCrypt/deobfuscation-and-beyond.

[5] J. Carette. Understanding Expression Simplification. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 72–79, New York, NY, USA, 2004. ACM.

[6] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, Aug. 2002.

[7] B. Dang, A. Gazet, E. Bachaalany, and S. Josse. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, chapter 5: Obfuscation. Wiley Publishing, 2014.

[8] L. de Moura and N. Bjørner. 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*, TACAS'08/ETAPS'08, pages 337–340. Springer, 2008. https://github.com/Z3Prover/z3.

[9] F. Desclaux. Miasm: Framework de reverse engineering. In *Actes du SSTIC*. SSTIC, 2012. https://github.com/cea-sec/miasm.

[10] Francis Gabriel. Deobfuscation: recovering an OLLVM-protected program. Quarkslab's blog, 2014. http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html.

[11] Y. X. Gu, C. Liem, and Y. Zhou. System and method providing dependency networks throughout applications for attack resistance. App. PCT/CA2011/050157, Publication Number WO2012126083 A1, Sept. 2012. Irdeto Canada Corporation.

[12] H. J. Johnson, Y. X. Gu, and Y. Zhou. System and method of interlocking to protect software-mediated program and device behaviors. US Patent App. 11/980,392, Publication Number US20080208560 A1, Aug. 2008.

[13] A. Kandanchatha and Y. Zhou. System and method for obscuring bit-wise and two's complement integer computations in software. US Patent App. 11/039,817,

Publication Number US20050166191 A1, Jul. 2005. Cloakware Corporation.

[14] D. Khovratovich and I. Nikolić. Rotational Cryptanalysis of ARX. In *Fast Software Encryption*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 2010.

[15] A. Klimov and A. Shamir. A New Class of Invertible Mappings. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 470–483. Springer, 2003.

[16] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[17] T. R. Kuphaldt. Lessons in Electric Circuits, Vol. IV - Digital, Chap. 7 - Boolean Algebra. http://www.allaboutcircuits.com/textbook/digital/#chpt-7, 1996.

[18] X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. In *Advances in Cryptology — EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 1991.

[19] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 36–52. Springer, 2008.

[20] Maple (Release 12.0). Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. http://www.maplesoft.com/.

[21] J. T. McDonald. Capturing the essence of practical obfuscation. In *Proceedings of the 6th International Conference on Information Systems, Technology and Management, ICISTM 2012*, volume 285 of *Communications in Computer and Information Science*, pages 451–456. Springer, 2012.

[22] C. Mougey and F. Gabriel. DRM obfuscation versus auxiliary attacks. Recon conference, 2014. https://recon.cx/2014/schedule/events/44.html.

[23] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0. *JSAT*, 9:53–58, 2015.

[24] D. Plump. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, chapter Term Graph Rewriting, pages 3–61. World Scientific Publishing Co., Inc., 1999.

[25] R. M. Stallman and the GCC Developer Community. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[26] F. Saudel and J. Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[27] J. Vuillemin. Digital Algebra and Circuits. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 733–746. Springer, 2003.

[28] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[29] I. Wegener. *The Complexity of Boolean Functions*. John Wiley & Sons, Inc., New York, NY, USA, 1987.

[30] Y. Zhou and A. Main. Diversity Via Code Transformations: A Solution For NGNA Renewable Security. Technical report, The NCTA Technical Papers, 2006.

[31] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *8th International Workshop in Information Security Applications (WISA'07)*, pages 61–75, 2007.