Effectiveness of Synthesis in Concolic Deobfuscation

Fabrizio Biondi[a], Sébastien Josse[b], Axel Legay[a], Thomas Sirvent[b]

[a]*IRISA/Inria Rennes, 263 Avenue du Général Leclerc, 35042 Rennes, France*
[b]*DGA Maîtrise de l'information, Rue Champion de Cice, 35170 Bruz, France*

Abstract

Control flow obfuscation techniques can be used to hinder software reverse-engineering. Symbolic analysis can counteract these techniques, but only if they can analyze obfuscated conditional statements. We evaluate the use of dynamic synthesis to complement symbolic analysis in the analysis of obfuscated conditionals. We test this approach on the taint-analysis-resistant Mixed Boolean Arithmetics (MBA) obfuscation method that is commonly used to obfuscate and randomly diversify statements. We experimentally ascertain the practical feasibility of MBA obfuscation. We study using SMT-based approaches with different state-of-the-art SMT solvers to counteract MBA obfuscation, and we show how targeted algebraic simplification can greatly reduce the analysis time. We show that synthesis-based deobfuscation is more effective than current SMT-based deobfuscation algorithms, thus proposing a synthesis-based attacker model to complement existing attacker models.

1. Introduction

Software obfuscation methods are extensively applied in various areas of digital right management (software protection, diversification, watermarking).

Opaque predicates are operations that have been rewritten in an equivalent but more complex form to hinder reverse engineering. Opaque predicates are used in obfuscation techniques to add unfeasible paths, thus increasing the complexity of deobfuscation and recognition. Many off-the-shelf compilation chains line Epona [1] and LLVM Obfuscator [2] currently implement opaque predicates. This method, used in conjunction with other techniques, such as *control flow flattening* and *virtualization-based obfuscation*, make the task harder for reverse-engineers.

Control flow flattening replaces the control flow logic with a dispatch-execute loop, forcing an adversary to perform global analysis to understand local control flow transfers and obstructing both forward and backward analysis. Virtualization-based obfuscation translates parts of the source code to be obfuscated into a

Email addresses: `fabrizio.biondi@inria.fr` (Fabrizio Biondi), `sebastien.josse@polytechnique.edu` (Sébastien Josse), `axel.legay@inria.fr` (Axel Legay), `thomas.sirvent@m4x.org` (Thomas Sirvent)

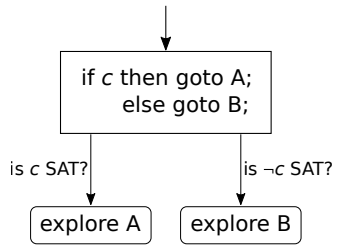


Figure 1. Symbolic execution. When a conditional statement is found the guard c and its negation are tested for satisfiability, and the two branches are explored accordingly.

byte-code representation that is executed by an embedded virtual machine, possibly with a randomly generated instruction set. Such randomized obfuscation makes it possible to diversify the binary generation process. Virtualization is implemented in off-the-shelf obfuscation programs like Themida [3], Code virtualizer [4], VMProtect [5], and Tigress [6].

State of the art in deobfuscation shows that control flow flattening not based on opaque predicates can be broken by using static path deobfuscation [7]. Recent work [8, 9] focuses on the use of symbolic analysis together with taint analysis to deobfuscate virtualized binaries and allow exploration of their execution path. Symbolic analysis maintains sets of constraints on the execution paths to determine which inputs cause each branch of a conditional statement to be explored. *Concolic* analysis combines concrete execution of program traces with symbolic analysis to increase code coverage and trigger hidden behavior. This depends on the ability to *symbolically determine the satisfiability of conditional statements*, since the analyzer has to decide which branches of the conditional to analyze, as shown in Fig. 1.

Control flow flattening transformations can be reinforced by implementing the dispatcher with cryptographic hash functions and opaque predicates [10], and in this scenario, it has been shown that statically breaking this obfuscation transformation depends on the ability to statically analyze the opaque predicates. The seed of the hash function is a vector of opaque predicates itself. The challenge for the obfuscator is how to prevent the cryptographic hash function and its seed to be easily detected. Diversification techniques based on *white-box cryptography* can be used to hide the signature of the hash function and its seed.

Several white box implementations of well-known block ciphers have been cryptoanalyzed using black-box methods [11, 12]. Inspired by this, we want to test the effectiveness of black-box dynamic synthesis to break or simplify opaque predicates. Dynamic synthesis interrogates a program by considering it as a black box and inductively synthesizes it by learning from its input/output behavior. We *show how dynamic synthesis can improve concolic analysis by simplifying constraints*. We used dynamic synthesis to implement a cryptanalysis method that produces an equivalent, concise form in Algebraic Normal Form (ANF) of the obfuscated conditionals. This form simplifies the constraints handled by the constraint solver.

Since the white-box representation of a cryptographic hash function can be the core component of an obfuscated conditional, in this work we will focus on the analysis of obfuscated conditionals. In this sense, obfuscated conditionals correspond to contextual opaque predicates [13] in the sense that they can be true or false according to the constraints under which they are evaluated. To illustrate the immediate utility of our approach, we will consider the Mixed Boolean Arithmetics (MBA) obfuscation technique. Companies using MBA obfuscation in their products include Quarkslab [1] and Irdeto [14]. MBA obfuscation is resistant to taint analysis because it systematically induces a massive overtainting, even if bit-level tainting is considered.

MBA obfuscation is used to both obfuscate and diversify a conditional statement. We observe that its robustness has never been assessed in the literature, so one of the contributions of this paper is to evaluate it.

Several attacker models have proved to be efficient against obfuscated binaries. Each one targets a specific class of obfuscation mechanisms. Taint-based analysis, dynamic binary translation combined with control dependencies and optimization transformations are efficient, e.g., against virtualization-based obfuscation. Our synthesis-based approach can be considered a different, complementary attacker model to the ones listed above.

Contributions. This work provides the following contributions:

1. We perform an experimental feasibility analysis of MBA obfuscation, in terms of obfuscation time and size of the obfuscated code. We show how the time required for obfuscation and the obfuscated file size grow exponentially with the degree of the polynomial used for the obfuscation.
2. We test the effectiveness of SMT-based techniques for determining the truth values of MBA-obfuscated conditional statements. We find that SMT solvers can solve MBA-obfuscated predicates in a time in the order of tens of seconds.
3. We present an algebraic simplification approach that reduces the complexity of polynomial MBA deobfuscation to the deobfuscation of linear MBA obfuscation. We show that the algebraic simplification is orders of magnitude faster than SMT solvers. However, it only works if the MBA follows a specific construction that is easy to change, thus is not general enough to be considered as an efficient technique in practice.
4. We test a dynamic synthesis method for determining the truth values of MBA-obfuscated conditional statements, and show its higher effectiveness compared to SMT-based techniques. Since synthesis is also more general than algebraic simplification, we conclude that synthesis is the most effective method among the ones we have evaluated.

Outline. The rest of this paper is structured as follows: Section 2 introduces MBA obfuscation, the drill-and-join synthesis method, and the mathematical theory necessary to understand them. Section 3 discusses the practical feasibility of MBA obfuscation, determining bounds on the degree of the polynomials that

the technique can use in practice. Section 4 tests the effectiveness of SMT solvers in determining the truth value of obfuscated conditionals and Section 5 gives an algebraic simplification technique to greatly reduce the deobfuscation time. Section 6 evaluates the drill-and-join synthesis method on the same problem. Section 7 discusses related work, while Section 8 concludes the paper and points out possible future research directions.

2. Background

We introduce some basic concepts that will be used in the rest of the paper.

2.1. Mathematical Background

Let us denote by $\mathbb{Z}_{2^n}$ the quotient ring of integers modulo 2^n and by $\mathbb{F}_2^n$ the ring $(\mathbb{Z}_2)^n$ of $n-$tuples of elements in the quotient ring of integers modulo 2 $(\mathbb{Z}_2)$.

2.1.1. Boolean Arithmetic Algebra

Let $n \in \mathbb{N}_0$ and $\mathrm{B} = \{0, 1\}$. Assume the following operations over the integer modular ring $\mathbb{Z}_{2^n}$: addition $+$, subtraction $-$, multiplication $\cdot$, comparison $<, \leq, =, \geq, >$, signed comparison $<^s, \leq^s, \geq^s, >^s$, left shift $\ll$, logical right shift $\gg$, arithmetic right shift $\gg^s$, conjunction $\wedge$, disjunction $\vee$, exclusive disjunction $\oplus$, and negation $\neg$. Then we define a *Boolean arithmetic algebra (BA-algebra)* as follows:

Definition 1. *[15] The algebraic system*

$$\mathrm{BA}[n] = (\mathrm{B}^n, \wedge, \vee, \oplus, \neg, <, \leq, =, \geq, >, <^s, \leq^s, \geq^s, >^s, +, \cdot)$$

is a Boolean-arithmetic algebra of dimension n.

Let $t \in \mathbb{N}_0$ and let $I, J_i \subset \mathbb{Z}$ be finite index sets for all $i \in I$. Assume constants α_i and bitwise expressions $e_{i,j}$ of variables $x_1, \ldots, x_t$ over B^n for $j \in J_i$. Then we define a *polynomial mixed Boolean arithmetic (MBA) expression* as follows:

Definition 2. *[15] A function $f : (\mathrm{B}^n)^t \mapsto \mathrm{B}^n$ of the form*

$$\sum_{i \in I} \alpha_i \left(\prod_{j \in J_i} e_{i,j}(x_1, \ldots, x_t) \right)$$

is a polynomial mixed Boolean-arithmetic expression.

We say that a polynomial MBA expression is *linear* if it is in the form

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

2.1.2. Synthesis Methods

A dynamic synthesis method is used to inductively synthesize a target program by learning from its input/output behavior. The target program is considered as a black box oracle and interrogated by the synthesizer, which constructs a function simulating the behavior of the oracle with the highest possible precision.

Let $\mathbb{F}_2^n$ be the set of all n-tuples of elements in the Galois field $\mathbb{F}_2$. Any vectorial Boolean function $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$ can be represented by the vector of its component functions $(f_1, f_2, \ldots, f_m)$ where each f_i is a Boolean function $f_i : \mathbb{F}_2^n \to \mathbb{F}_2$.

The general idea of vectorial Boolean function synthesis is to use a *divide and conquer* approach: first, find a way to synthesize each component f_i for $i = 1, \ldots, m$ independently, potentially using parallel processing. Then the components are combined to produce the target function F.

The synthesis of each component Boolean function f_i follows the *expansion* approach, in which the target function is iteratively separated in its own subspaces, with each division decreasing the dimension of the vectorial space of the function by at least 1, until the bases of the function are found and recombined in the function f_i itself. The drill-and-join synthesis method we consider in this paper has been recently introduced by Balaniuk [16] as an efficient implementation of this idea.

We will consider each component function f_i to be in Algebraic Normal Form (ANF). We introduce ANF in more details in Section 2.1.3.

2.1.3. Algebraic Normal Form

Algebraic Normal Form (ANF), also known as positive polarity Reed-Muller form, is a compact form for the representation of Boolean functions. A formula in ANF is an exclusive disjunction of clauses, where each clause is a conjunction of variables. The negation operator is unnecessary, as all variables appear in positive form. More formally, the ANF of a Boolean function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ is

$$f(\bar{x}) = \bigoplus_{\bar{j} \in \mathbb{F}_2} a_{\bar{j}} \bar{x}^{\bar{j}}$$

where $a_{\bar{j}} \in \mathbb{F}_2$, $\bar{x} = (x_0, x_1, \ldots, x_{n-1})$, $\bar{j} = (j_0, j_1, \ldots, j_{n-1})$ and $\bar{x}^{\bar{j}} = (x_0^{j_0}, x_1^{j_1}, \ldots, x_{n-1}^{j_{n-1}})$. We denote by $\deg(f)$ the *degree* of f, i.e. the number of variables in the longest clause of the ANF of f. We call f an *affine function* iff $\deg(f) \leqslant 1$, and a *constant function* iff $\deg(f) = 0$.

2.1.4. Reed-Muller Expansion

Given a function $f : \mathbb{F}_2^n \to \mathbb{F}_2$, let

$$f_{x_i}(\bar{x}) = f(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_{n-1}) \ ,$$

$$f_{\neg x_i}(\bar{x}) = f(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_{n-1}) \ , \text{ and}$$

$$f'_{x_i}(\bar{x}) = f_{x_i}(\bar{x}) \oplus f_{\neg x_i}(\bar{x}) \ .$$

Then it holds that

$$f(\bar{x}) = f_{\neg x_i}(\bar{x}) \oplus x_i f'_{x_i}(\bar{x})$$

where the right term of the equation is known as Reed-Muller expansion (or positive Davio expansion) of f.

Each element in the expansion is a function belonging to a vector space of a lower dimension than the vector space of f. The expansion can be applied again to obtain more elements of an even lower dimension until they become all bit constants, and oracle-based synthesis uses calls to the black-box oracle to understand the values of such constants for the function under analysis and synthesize it. In general this requires a number of expansions linear in n, creating an exponential number of elements.

Advanced techniques like the drill-and-join algorithm presented in Section 2.3 follow a similar approach in an optimized way to find bases of the target function and efficiently synthesize a functional program with the same behavior as the target function. Still, the worst-case complexity for these techniques remains exponential in n.

2.2. MBA Obfuscation

We introduce the obfuscation method based on mixed Boolean arithmetics proposed by Zhou et al. [15]. The method is based on mixed mode computation over Boolean-arithmetic algebras and on invertible polynomial functions over the ring $\mathbb{Z}_{2^n}$. Both the polynomial and its inverse must be of limited degree so that polynomial code transformations are efficient, as we show in Section 3. The degree of the polynomial can be considered as a parameter chosen by the obfuscator. In the next sections we will analyze the impact of the degree chosen on the effectiveness of the obfuscation.

2.2.1. Obfuscation with a Polynomial

Consider a function that we want to obfuscate, for instance some constant key function k. MBA obfuscation is based on constructing a polynomial in m variables $x_1, \ldots, x_m$ that is equivalent to k, and substituting the polynomial to k in the code. The polynomial can be made very large by increasing its degree, thus making it harder for a symbolic execution method to analyze it.

Assume an invertible function f_d of degree d and a m-variables linear MBA identity $\sum_{i \in I} \alpha_i e_i = 0$ for some finite index set I. Then

$$k = f_d^{-1}(f_d(k)) = f_d^{-1}\left(\sum_{i \in I} \alpha_i e_i + f_d(k)\right)$$

and rearranging the terms of last element in the equation gives us the desired polynomial.

Details on how to construct an unlimited number of non-trivial linear MBA identities can be found in [15]. The degree d of the invertible function f_d is one of the parameters of the obfuscation process: the degree is chosen by the user and an appropriate invertible function f_d and its inverse f_d^{-1} is constructed via the following theorem:

Theorem 1. *[15, Theorem 3] Let $P_d(\mathbb{Z}_{2^n})$ be a set of polynomials over $\mathbb{Z}_{2^n}$:*

$$P_d(\mathbb{Z}_{2^n}) = \left\{ \sum_{i=0}^{d} a_i x^i \;\middle|\; \forall a_i \in \mathbb{Z}_{2^n}.\ a_1 \wedge 1 = 1,\ a_i^2 = 0,\ i = 2, \ldots, d \right\} \quad .$$

Then $(P_d(\mathbb{Z}_{2^n}), \circ)$ is a permutation group under the functional composition operator $\circ$. For every element $f_d(x) = \sum_{i=0}^{d} a_i x^i$, its inverse $f_d^{-1}(x) = \sum_{j=0}^{d} b_j x^j$ can be computed by

$$\forall k \geqslant 2,\ b_k = -a_1^{-k-1} a_k - a_1^{-1} \sum_{j=k+1}^{d} \binom{j}{k} a_0^{j-k} A_j,$$

$$b_1 = a_1^{-1} - a_1^{-1} \sum_{j=2}^{d} j a_0^{j-1} A_j$$

$$b_0 = -\sum_{j=1}^{d} a_0^j b_j$$

where $A_d = -a_1^{-d} a_d$, and A_k for $2 \leqslant k < d$ is recursively defined by

$$A_k = -a_1^{-k} a_k - \sum_{j=k+1}^{d} \binom{j}{k} a_0^{j-k} A_j \quad .$$

2.2.2. Example: Comparison of Variable and Constant

We introduce a simple conditional statement that will be used frequently in the rest of the paper: a comparison between a variable and a constant, proposed as an example in [15, Appendix A]. We will obfuscate this conditional statement with the MBA obfuscation method described above.

For this example we work in $\mathbb{Z}_{2^{32}}$. Assume that the conditional to be obfuscated compares some input IN with the constant $k = 0\text{x}87654321$:

```
if (IN == 0x87654321 )
..
else
..
```

We use the method described above to produce the two linear MBA identities:

$$E_1(x, y) \triangleq 2y = -2(x \vee (-y - 1)) - ((-2x - 1) \vee (-2y - 1)) - 3$$

$$E_2(x, y) \triangleq x + y = (x \oplus y) - ((-2x - 1) \vee (-2y - 1)) - 1$$

and one invertible polynomial transform of degree 2 respecting the conditions of Theorem 1:

$$f(x) \triangleq 727318528x^2 + 3506639707x + 6132886 \quad .$$

By adding three spurious input variables x, x_1 and x_2 the following obfuscated conditional is generated:

```
a = x * (x1 | 3749240069);
b = x * ((-2*x1 - 1) | 3203512843);
c = ((235810187 * x + 281909696 - x2)
     ^ (2424056794 + x2));
d = ((3823346922 * x + 3731147903 + 2 * x2)
     | (3741821003 + 4294967294 * x2));

f = 2284837645 + 272908530*a + 136454265*b
     + 409362795*x + 135832444*c
     + 4159134852*d + 415760384*a*a
     + 415760384*a*b + 1247281152*a*x
     + 2816475136*a*c + 1478492160*a*d
     + 3325165568*b*b + 2771124224*b*x
     + 1408237568*b*c + 2886729728*b*d
     + 4156686336*x*x + 4224712704*x*c
     + 70254592*x*d + 1428160512*c*c
     + 1438646272*c*d + 1428160512*d*d;
if (IN == f )
..
else
..
```

The output value of f is always the constant $k = 0\text{x}87654321$ regardless of values in x, x_1 and x_2 because it corresponds to:

$$\begin{aligned} & f^{-1}(1723234813\,x\,E_1' + 294146324\,E_2' + f(k)) \\ & = f^{-1}(f(k)) = k, \end{aligned}$$

since $E_1' = E_1(x_1, 545727226) - 1091454452$, and $E_2' = E_2(235810187x + 281909696 - x_2, 2424056794 + x_2) - 235810187x - 2705966490$ are two linear MBA identities, i.e., are null.

This simple example is useful to understand MBA obfuscation in practice, and has been used as the basis of some of our experiments including the ones in Section 3. However, the SMT-based and synthesis-based deobfuscation methods presented in Sections 4 and 6 respectively are not limited to this example.

2.3. Drill-and-Join Synthesis

The drill-and-join method was recently proposed as a new inductive method for efficient program synthesis [16].

Let us recall the principle of this method. Let $x|y$ denote the concatenation of x and y. Let $(v_1, \ldots, v_m) \in \mathbb{F}_2^n$ be a basis of the $\mathbb{F}_2$-vector space $\mathcal{F} = span(v_1, \ldots, v_m)$ and define $dim(\mathcal{F})$ as the minimum m such that

$$\mathcal{F} = \{\bigoplus_{i=1}^{m} \lambda_i.v_i \,|\, \lambda_1, \ldots, \lambda_m \in \mathbb{F}_2\} \quad .$$

The method uses the two maps δ (drill) and γ (join), each of which is related to an expansion formula as shown below. The algorithm aims to synthesize

separately each component function f_i of $F = (f_1, f_2, \ldots, f_m)$, each one of which computes one of the output bits of F. The synthesis of each component function is independent from each other and can be computed in parallel.

For each component function f_i, the algorithm works by recursively reducing the dimension of the vector space the function belongs to, until such dimension becomes zero. If the algorithm has a basis for the subspace it is working on, it uses the join map to reduce the dimension of the vector space by at least 1 and calls itself again. If it does not have a basis it calls the drill map to reduce the dimension of the vector space instead, and then calls itself again. The drill map does not require a basis for the subspace, but has to evaluate three functions in the reduced subspace, leading to a potential exponential increase in the number of functions evaluated. For this reason the algorithm uses join whenever a basis is available, and drill otherwise.

The effectiveness of the algorithm can be improved by using a cache of the bases of subspaces, reducing the number of subqueries to the black-box oracle. However, we have not exploited such capability in our experiments.

We now present the drill and join maps in more detail.

The *drill* map δ is used to produce the bases of a functional program simulating a given target function by dividing the function's space in subspaces and recursively calling itself on them:

Definition 3. *Given a boolean function* $f : \mathbb{F}_2^n \to \mathbb{F}_2$*, if* $f(x_0|y_0) = 1$ *then the* drill *function* δ *maps* $f \in \mathcal{F}_m \mapsto \delta f \in \mathcal{F}_r$ *with* $r = dim(\mathcal{F}_r) \leq m - 1$ *such that*

$$f(x|y) \quad = \quad \delta f(x|y) \oplus (f(x_0|y) \wedge f(x|y_0)) \quad .$$

The required x_0 and y_0 for which $f(x_0|y_0) = 1$ are determined by querying the black-box oracle until suitable values are found. The δ map can be applied recursively, each recursion generating a function belonging to a vector space of lower dimension:

$$\begin{aligned} f(x|y) &= \delta^1 f(x|y) \oplus (f(x_0|y) \wedge f(x|y_0)) \\ &= \delta^2 f(x|y) \oplus \left(\delta^1 f(x_1|y) \wedge \delta^1 f(x|y_1)\right) \\ &\qquad \oplus (f(x_0|y) \wedge f(x|y_0)) \\ &= \delta^m f(x|y) \oplus \bigoplus_{i=1}^{m-1} \left(\delta^i f(x_i|y) \wedge \delta^i f(x|y_i)\right) \\ &\qquad \oplus (f(x_0|y) \wedge f(x|y_0)) \\ &= \bigoplus_{i=0}^{m-1} \left(\delta^i f(x_i|y) \wedge \delta^i f(x|y_i)\right) \quad . \end{aligned}$$

The recursion ends with the zero-map $\delta^m = 0$, whose value is verified by oracle query.

The *join* map γ is used to find the relevant basis to express an element of a given vector space:

Definition 4. *Given a basis* $(w_i)_{1 \leq i \leq m}$*, if* $\exists z_0 \in \mathbb{F}_2^n$ *and* $\exists v_0 \in (w_i)_{1 \leq i \leq m}$ *such that* $f(z_0) = 1$ *and* $v_0(z_0) = 1$*, then the* join *map* γ *maps* $f \in \mathcal{F}_m \mapsto \gamma f \in \mathcal{F}_r$

with $r = dim(\mathcal{F}_r) \leq m-1$ *such that*

$$f(z) \quad = \quad \gamma f(z) \oplus v_0(z) \quad .$$

The required z_0 such that $f(z_0) = 1$ is determined by querying the black-box oracle until a suitable value is found. The γ map can be applied recursively, each recursion generating a function belonging to a vector space of lower dimension :

$$\begin{aligned} f(z) &= \gamma^1 f(z) \oplus v_0(z) \\ &= \gamma^2 f(z) \oplus v_1(z) \oplus v_0(z) \\ &= \gamma^m f(z) \oplus \bigoplus_{i=0}^{m-1} v_i(z) \\ &= \bigoplus_{i=0}^{m-1} v_i(z) \quad . \end{aligned}$$

The recursion ends with the zero-map $\gamma^m = 0$, whose value is verified by oracle query.

In Section 6 we present the results we have obtained by using the drill-and-join method to counteract MBA obfuscation.

2.3.1. Example

We present a simple example of the drill-and-join algorithm in action [16]. Consider the target function $f(x|y) = y \vee (\neg x \wedge \neg y)$.

$$\begin{aligned} f(x|y) &= \delta^1 f(x|y) \oplus (f(0|y) \wedge f(x|0)) && (i) \\ &= \delta^2 f(x|y) \oplus (\delta f(1|y) \wedge \delta f(x|1)) \oplus (f(0|y) \wedge f(x|0)) && (ii) \\ &= x \wedge y \oplus x \oplus 1 && (iii) \end{aligned}$$

We apply the drill function with $x_0 = y_0 = 0$ and obtain two subspace problems $f(0|y)$ and $f(x|0)$ (i). Since $\delta^1 f$ is not the zero map we apply drill recursively with $x_1 = y_1 = 1$, obtaining the subspace problems $\delta^1 f(1|y)$ and $\delta^1 f(x|1)$. Since $\delta^2 f$ is the zero map, the recursion terminates (ii).

The join function is used to solve the four obtained one-dimensional subspace problems $f(0|y)$, $f(x|0)$, $\delta^1 f(1|y)$, and $\delta^1 f(x|1)$ using basis $v_0(z) = 1, v_1(z) = z$ (where z represents x or y). Consider the problem $f(z) := f(x|0)$. We apply the join function recursively two times with $z_0 = 0, v_0(z) = 1$ and $z_1 = 1, v_1(z) = z$ respectively to obtain the zero map, so we reconstruct it as $f(z) = v_0(z) \oplus v_1(z) = z \oplus 1$:

$$\begin{aligned} f(z) &= \gamma^1 f(z) \oplus v_0(z) && (z_0 = 0, f(z_0) = 1, v_0(z_0) = 1) \\ &= \gamma^2 f(f) \oplus v_1(z_1) \oplus 1 && (z_1 = 1, \gamma^1 f(z_1) = 1, v_1(z_1) = z_1 = 1) \\ &= z \oplus 1 \end{aligned}$$

Similarly we obtain $f(0|y) = 1$, $\delta^1 f(1|y) = y$, and $\delta^1 f(x|1) = x$.

Finally, we obtain (iii) the target function as $f(x|y) = (f(0|y) \wedge f(x|0)) \oplus (\delta^1 f(1|y) \wedge \delta^1 f(x|1)) = (1 \wedge (x \oplus 1)) \oplus (y \wedge x) = x \wedge y \oplus x \oplus 1$ which is the equivalent algebraic normal form of our initial representation $f(x|y) = y \vee (\neg x \wedge \neg y)$.

3. Feasibility of MBA Obfuscation

3.1. Cost of MBA Obfuscation

The MBA obfuscation method replaces simple statements with longer equivalent statements, as explained in Section 2.2. In this section we evaluate the size and execution overhead of obfuscated code. The results are depicted in Figure 2. All experiments in this and in the following sections are conducted on a Intel Core i5 1.60-2.30 GHz.

The red line with dots on the left in Figure 2 represents the size of the obfuscated binary file corresponding to the obfuscated constant comparison statement from Section 2.2.2. The graph shows that the size of the compiled binary grows exponentially with the degree of the polynomial used. The trend shows that using high-degree polynomials has a considerable impact on the size of the obfuscated binary. For instance, using a degree 5 polynomial means that every single obfuscated conditional statement in the source code occupies ~ 370 kB, making the total size of the binary program impractically large.

The blue line with xs on the right in Figure 2 represents the increase in execution time of the obfuscated constant comparison statement from Section 2.2.2 compared to its unobfuscated equivalent statement. As the graph shows, obfuscation with a polynomial of degree 2 to 4 does not increase significantly the execution time, while using a polynomial of degree 5 increases it by 40%, a polynomial of degree 6 almost doubles it, and a polynomial of degree 7 increases it by 14 times. The trend shows that using high-degree polynomials has a considerable impact on the execution time of the obfuscated system: the execution time ratio grows quickly.

We conclude that *the MBA obfuscation method becomes impractical when using polynomials of high degree*, due to the time required to produce the obfuscated statements and the increase in execution time. For this reason, in the rest of the paper we will not consider MBA obfuscation with polynomials of degree above 5 or 6.

3.2. MBA Obfuscation Detection

Locating opaque predicates in a program is not obvious, even for invariant opaque predicates like obfuscated constants. This is the goal of tools like the LOOP tool [17], that detects opaque predicates in a program and tries to break them using SMT solvers.

Assume that an attacker suspects that some compiled code contains an obfuscated constant, and wants to determine if the constant has been obfuscated using the MBA obfuscation method described above. Then the attacker will scan the code looking for evidence that the code has been produced by the obfuscation method.

The method builds multivariate polynomials similar to the expression of `f` in the obfuscated code in Section 2.2. The polynomial used has degree 2. The size of the polynomial grows quickly with its degree, so for obfuscation with higher-degree polynomials the polynomial itself becomes very visible and easy to find in the code.

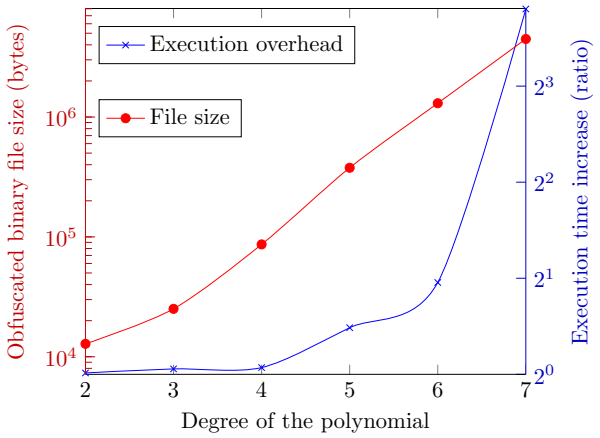


Figure 2. Cost of MBA obfuscation of the constant comparison case study in Section 2.2.2 using obfuscation polynomials of different degrees. Red line with dots on the left: obfuscated binary size in bytes. Blue line with xs on the right: Execution time increase for MBA obfuscation as a ratio of the execution time of unobfuscated code.

Once the attacker has identified a suspicious multivariate polynomial in the code, he can evaluate it with different values of the variables $x_1, \ldots, x_t$. This is the basis for the dynamic synthesis approach we present in Section 6.

4. SMT Solver-based Deobfuscation

Several symbolic analysis techniques use SMT solvers to determine whether a given conditional statement can be satisfied, considering the constraints that the analyzer has accumulated up to that point.

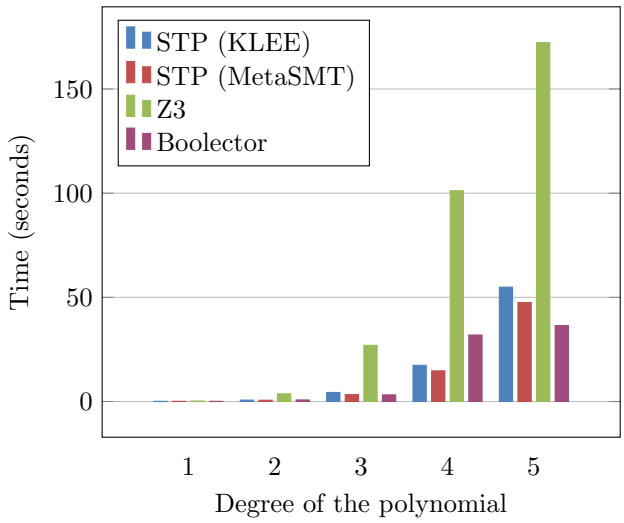
In this section we evaluate the effectiveness of several SMT solvers in deciding the satisfiability of MBA obfuscated conditionals.

4.1. Experimental Setup

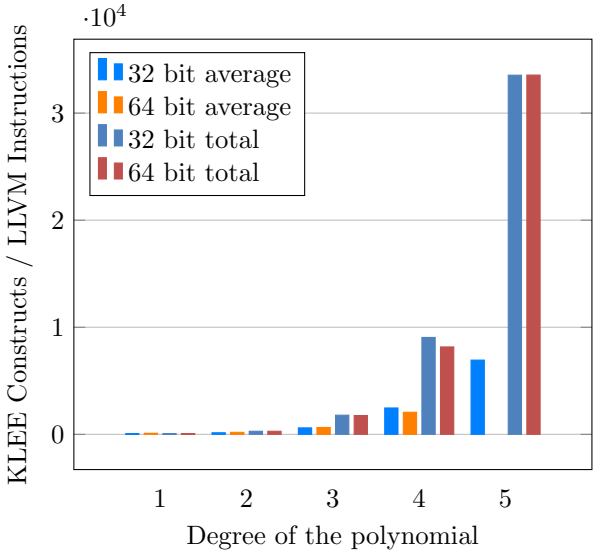
We test different SMT solvers on deciding satisfiability of an opaque predicate generated by applying MBA obfuscation with polynomial of different degrees to the comparison between an input and a constant described in Section 2.2.2.

We use the LLVM [18] compilation framework to implement the MBA obfuscation transformations. To evaluate the effectiveness of constraints solvers against MBA obfuscated conditionals, we use the KLEE symbolic execution engine [19] and the multi-solver support in KLEE provided by the MetaSMT framework [20].

KLEE expresses the SMT constraints in quantifier-free fix-size bit-vector logic with array, arbitrary solvers and function symbols (`QF_AUFBV`), thus we experiment with SMT solvers that are able to handle such logic.



(a) Analysis time to determine satisfiability of the obfuscated conditionals.



(b) Average number of KLEE constructs per query and total number of LLVM instructions

Figure 3. Satisfiability analysis of obfuscated conditionals with various SMT solvers using obfuscation polynomials of different degrees.

4.2. Solvers

We use the STP, Z3 and Boolector SMT solvers.

STP git-12/02/2015. The STP solver [21] supports queries in the CVC and SMT-LIB languages. It uses word-level preprocessing with several heuristics including array abstraction refinement and a bit-vector linear arithmetics equation solver, then it translates the words to SAT for satisfiability checking. STP is the default SMT solver used internally by KLEE, but it is also supported by the MetaSMT framework. We used both versions of STP to verify whether MetaSMT has an impact on the resolution time.

Z3 4.1. Z3 [22] handles statements in the SMT-LIB language. It uses a DPLL-based SAT solver, a core theory solver handling equalities and uninterpreted functions, and integrates its functionalities with satellite theory solvers for linear arithmetics, bit-vectors, arrays and other theories. The models maintained by each theory solver are combined incrementally. Quantifier are handled by an abstract machine for matching.

Boolector 1.5.118. Boolector [23] supports queries in its own language BTOR and in the SMT-LIB language. It uses strong rewriting algorithms to simplify the statements just after parsing, which we expect to be effective for the deobfuscation of obfuscated conditionals. It starts by checking satisfiability of an overapproximation of the formula of interest, and uses a counterexample-based iterated refinement approach to verify its satisfiability. It also supports bit-blasting for bit-vectors and lazy handling of the theory of arrays.

4.3. Results

Figure 3(a) gives the time (in seconds) required to analyze randomly generated MBA obfuscated constraints, for MBA expressions of 16 variables and polynomials of degree 2, 3, 4 and 5.

The data shows that Z3 is the slowest SMT solver for handling obfuscated conditionals, and that Boolector is slightly faster than STP on higher degree cases. The two different implementations of STP, integrated in KLEE and via metaSMT, do not differ significantly in solution time, showing that metaSMT does not add a significant overhead to the solvers.

The graph in Figure 3(a) shows that the MBA obfuscation method is effective in slowing down the analysis of conditional statements even when using a polynomial with a small degree. Therefore, MBA obfuscation is effective in slowing down control flow deobfuscation based on SMT techniques, i.e. when a large number of conditional statements have to be deobfuscated and the attacker is not able to spend the required hundreds of seconds for each one.

Figure 3(b) gives the average size of a solver query (measured by the number of constructs internally counted by KLEE) and the total number of LLVM instructions for MBA expressions of 16 variables and polynomials of degree 2, 3, 4 and 5 over integers of size 32 and 64 bits. Results for 64-bit average and degree 5 are missing due to a bug in the KLEE engine. The graph is helpful in

understanding the reason why SMT solvers are ineffective in addressing MBA conditional deobfuscation: the number of constructs expressing the conditional statement grows quickly with the degree of the polynomial used, and SMT solvers' computation time is sensitive to this number of constructs.

The graph also shows that the size n of the ring $\mathbb{Z}_{2^n}$ does not impact the number of constructs generated. We observe that the effectiveness of the obfuscation method is unaffected by the number of bits in the architecture targeted. The number of instructions grows with the polynomial's degree following Newton's multinomial formula. Let $d = 2, \ldots, 5$ be the degree of the polynomial and $m = 16$ the number of bit-vectors. Then the number of distinct m-tuples of non-negative integers whose sum is lesser or equal to d is

$$\binom{d+m}{d} = \frac{(d+m)!}{d!\,m!} \sim O(m^m + d^d).$$

To summarize, SMT solvers are not efficient enough to consider them a sufficient measure of deobfuscation for the concolic execution scenario. In Section 5 we present a technique to increase the effectiveness of SMT solvers against MBA obfuscation, based on simplifying the algebraic structure of the obfuscation. In Section 6 we explore the application of the more general drill-and-join synthesis algorithm to synthesize the obfuscated function instead of using SMT solvers to determine its satisfiability.

5. Algebraic Simplification

In this section we simplify MBA-obfuscated conditionals using computer algebra systems. We observe that even if computer algebra systems may encounter some problem to simplify the above formula (as mentioned by Zhou et al. in [15]), such a construction can be recognized, analyzed and simplified algebraically, thank to the specific form of the MBA-obfuscated conditional. Using this specific form, we can compute the polynomial on one hand, and the linear MBA identity on the other hand. From this separation, we solve the linear MBA identity, and then evaluate the polynomial to recover the obfuscated conditional.

This simplification provides a way to decide the satisfiability of an MBA-obfuscated conditional of any degree with the same complexity of deciding the satisfiability of an MBA-obfuscated conditional of degree 1.

5.1. Description of the Simplification Technique

The sketch of the technique is as follows. Given the MBA obfuscated polynomial, we recover immediately the expanded form of the following expression:

$$f_d^{-1}\left(\sum_{i\in I} \alpha_i e_i + f_d(k)\right).$$

We begin by finding precisely the expressions (e_i) used in this formula. This step is easy, due to their bitwise nature.

It is sufficient to find in the formula the expressions that appear only with a set of bitwise expressions, and with the same power.

Once identified, these expressions (e_i) are seen as variables of a multivariate polynomial g_d. Then we find the coefficients $(\alpha_i)_{i\in I}$ in the linear MBA identity, the coefficients $(b_j)_{0\leqslant j\leqslant d}$ of the polynomial f_d^{-1}, and the expression $f_d(k)$. Finally, we compute the value k hidden in the expression as $f_d^{-1}(f_d(k))$.

Now we describe in detail the steps outlined above. The reader uninterested by such details can skip to Section 5.2 for an example of the technique in action and further discussion.

5.1.1. Identification of the expressions: (e_i)

Given the MBA obfuscated polynomial, we recover immediately the expanded form of the following expression:

$$f_d^{-1}\left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right).$$

The first step consists in finding precisely the expressions (e_i) used in this formula. This step is easy, due to their bitwise nature. We remark that the expressions (e_i) can become more complex: this is the case of the example given in section 2.2.2 where the first linear identity is multiplied with a variable x. These multiplications can however be easily detected, as it is sufficient to find in the formula the expressions that appear only with a set of bitwise expressions, and with the same power.

Once identified, these expressions (e_i) are seen as variables of a multivariate polynomial g_d. Our goal is now to find the coefficients $(\alpha_i)_{i\in I}$ in the linear MBA identity, the coefficients $(b_j)_{0\leqslant j\leqslant d}$ of the polynomial f_d^{-1}, and the constant $f_d(k)$.

5.1.2. Equivalent decompositions: $\alpha_{i_0} = 1$

We observe that multiple solutions can be found for the (α_i) and the (b_j). All these equivalent decompositions are equally valid: each one of them allows the analysis and simplification of the linear MBA identity. We make here some assumptions on the decomposition that we will compute, and explain why these assumptions are justified.

We note that the construction of the polynomials f_d and f_d^{-1} is based on the following properties of the coefficients of f_d^{-1}: the coefficient b_1 of degree 1 is odd (invertible in $\mathbb{Z}_{2^n}$), while all other coefficients $(b_j)_{j\neq 1}$ are even.

We assume now that one of the coefficients α_i is invertible in $\mathbb{Z}_{2^n}$, i.e. odd, and we call α_{i_0} this coefficient. If this is not the case, all coefficients (except maybe the constant one) of the multivariate polynomial are even. We remove then the coefficient of degree 0, and we simply divide all of the remaining coef-

ficients by 2, ℓ times, until at least one of them becomes odd.

$$\begin{aligned} f_d^{-1}\left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right) &= \sum_{j=0}^{d} b_j \left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right)^j \\ &= \sum_{0\leqslant m\leqslant j\leqslant d} b_j \binom{j}{m}\left(\sum_{i\in I}\alpha_i e_i\right)^m (f_d(k))^{j-m} \\ &= \sum_{0\leqslant j\leqslant d} b_j\,(f_d(k))^j + 2^\ell\; g_d\left(\sum_{i\in I}\frac{\alpha_i}{2^\ell}\,e_i\right), \end{aligned}$$

where the polynomial g_d is defined with:

$$g_d(x) = \sum_{1\leqslant m\leqslant j\leqslant d} \binom{j}{m} b_j\, 2^{(m-1)\ell}\; (f_d(k))^{j-m}\; x^m.$$

The polynomial g_d has degree d, with coefficients in $\mathbb{Z}_{2^n}$. Its coefficient of degree 1 is odd, while all other coefficients are even (since b_1 is odd, while all other b_j are even). We can then identify the number ℓ of divisions by 2, and apply our method with g_d, instead of f_d^{-1}, where one coefficient at least is odd.

We assume now that this invertible coefficient α_{i_0} is 1. This is justified since we can divide all coefficients (α_i) and the constant $f_d(k)$ by α_{i_0}, and multiply all coefficients of f_d^{-1} by powers of α_{i_0}: the expansion remains the same, but the decomposition has a coefficient α_{i_0} equal to 1.

$$\begin{aligned} f_d^{-1}\left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right) &= \sum_{j=0}^{d} b_j \left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right)^j \\ &= \sum_{j=0}^{d} \left(b_j\,{\alpha_{i_0}}^j\right)\left(\sum_{i\in I}\left(\frac{\alpha_i}{\alpha_{i_0}}\right)e_i + \frac{f_d(k)}{\alpha_{i_0}}\right)^j. \end{aligned}$$

5.1.3. Identification of the linear MBA identity: (α_i)

For each $i \in I$, we denote by $\beta_{i,1}$ the coefficient of each expression e_i (of degree 1) in the expanded form of:

$$f_d^{-1}\left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right) = \sum_{j=0}^{d} b_j \left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right)^j.$$

We obtain the following relation:

$$\forall i \in I,\ \beta_{i,1} = \left(\sum_{j=1}^{d} j\, b_j\ (f_d(k))^{j-1}\right)\alpha_i.$$

We note that the coefficient of α_i in the expression of $\beta_{i,1}$ does not depend on i, and is invertible in $\mathbb{Z}_{2^n}$. We can thus compute all (α_i) from these coefficients $(\beta_{i,1})$:

- find an invertible coefficient $\beta_{i,1}$,
- define the index of this invertible $\beta_{i,1}$ as i_0,
- for all $i \in I$, define $\alpha_i = \beta_{i,1}/\beta_{i_0,1}$.

5.1.4. Equivalent decompositions: $f_d(k) = 0$

The coefficients (α_i) of the linear MBA identity are now known. We assume now that the constant $f_d(k)$ is null. As previously, we show here that an equivalent decomposition, with the same coefficients (α_i), respects this constraint.

$$\begin{aligned} f_d^{-1}\left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right) &= \sum_{j=0}^{d} b_j \left(\sum_{i\in I}\alpha_i e_i + f_d(k)\right)^j \\ &= \sum_{m=0}^{d}\left(\sum_{j=m}^{d}\binom{j}{m} b_j\ (f_d(k))^{j-m}\right)\left(\sum_{i\in I}\alpha_i e_i\right)^m. \end{aligned}$$

5.1.5. Identification of the polynomial f_d^{-1}: (b_j)

For each $j \in \{0,\ldots,d\}$, we denote by $\beta_{i_0,j}$ the coefficient of $(e_{i_0})^j$ (of degree j) in the expanded form of:

$$f_d^{-1}\left(\sum_{i\in I}\alpha_i e_i\right) = \sum_{j=0}^{d} b_j \left(\sum_{i\in I}\alpha_i e_i\right)^j.$$

Since $\alpha_{i_0} = 1$, we have the following relation:

$$\forall j \in \{0,\ldots,d\},\ \beta_{i_0,j} = b_j\,(\alpha_{i_0})^j = b_j.$$

The coefficients (b_j) of the polynomial f_d^{-1} are thus simply these coefficients $(\beta_{i_0,j})$.

5.1.6. Validation and answer

We need now to check that our decomposition is consistent with the original full expression given, since we only used a small number of its coefficients. This check is easy, since we only need to expand our decomposition, and control each coefficient.

We have then to control that the linear MBA identity found is constant. The original linear MBA expression is an identity, i.e. null, but our decomposition method can not find the constant part of this linear MBA identity: we expect thus a constant, not necessarily null. This verification can be performed in different ways. We can use synthesis-based deobfuscation (with low degree) as described in Section 6. An alternative way is to use the generation algorithm

given for such linear MBA identities: the linear MBA identities given in [15] are expansions of bit-based equations, that can be checked.

If our decomposition holds, and if the linear MBA expression is a constant c, then the value of the full expression is

$$f_d^{-1}(c) = \sum_{j=0}^{d} b_j \, c^j.$$

5.2. Example

We use the example given in 2.2.2. In the first step, we identify the follwing expressions: a, b, x, d and e. Their coefficients of degree 1 are: $\beta_{1,a} = 272908530$, $\beta_{1,b} = 136454265$, $\beta_{1,x} = 409362795$, $\beta_{1,d} = 135832444$, $\beta_{1,e} = 4159134852$.

The first invertible coefficient in $\mathbb{Z}_{2^n}$ in this list is $\beta_{1,b}$, and its inverse is: $1/\beta_{1,b} = 1761757641$. We can thus compute the linear MBA expression:

$$\begin{aligned}
\alpha_a &= \beta_{1,a}/\beta_{1,b} &&= 2,\\
\alpha_b &= \beta_{1,b}/\beta_{1,b} &&= 1,\\
\alpha_x &= \beta_{1,x}/\beta_{1,b} &&= 3,\\
\alpha_d &= \beta_{1,d}/\beta_{1,b} &&= 2230237276,\\
\alpha_e &= \beta_{1,e}/\beta_{1,b} &&= 2064730020 = -2230237276.
\end{aligned}$$

The coefficients of the polynomial f_d^{-1} are read as coefficients of powers of the variable b in the formula:

$$\begin{aligned}
b_0 &= \beta_0 &&= 2284837645,\\
b_1 &= \beta_{1,b} &&= 136454265,\\
b_2 &= \beta_{2,b^2} &&= 3325165568.
\end{aligned}$$

We check that the expansion of the decomposition corresponds to the full expression given. We check then that $\alpha_a \, a(x, x_1) + \alpha_b \, b(x, x_1) + \alpha_x \, x + \alpha_d \, d(x, x_2) + \alpha_e \, e(x, x_2)$ is constant, i.e. independent from the values of x, x_1 and x_2: this is really the case, and this constant is $c = 2661575604$.

We finally compute the value hidden in the full expression: $b_0 + b_1 \, c + b_2 \, c^2 = 2271560481 = 0\text{x}87654321$.

5.3. Complexity

The computation of the linear MBA identity is linear in the number of terms in this MBA identity. The identification of the polynomial f_d^{-1} is linear in the degree d of the polynomial, and the computation of the formula hidden in the full expression has a similar complexity. The most complex part is the verification. The verification of the linear MBA expression as a constant is not detailed here: it may be quite difficult but remains much easier than dealing with the original MBA expression of degree d. The verification of the decomposition is linear in the size of the full expression.

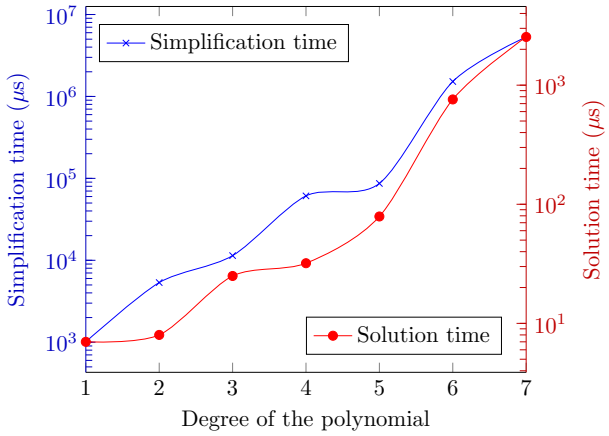


Figure 4. Algebraic simplification time for obfuscation polynomials different degrees. Blue line with xs on the left: Simplification time in microseconds. Red line with dots on the right: Solution time in microseconds.

5.4. Results

We have implemented the algebraic simplification method in C and applied it to the deobfuscation of the comparison between an input and a constant described in Section 2.2.2, obfuscated with MBA obfuscation using polynomial of different degrees. The time required for the algebraic simplification is presented in Figure 4. In the figure we show the time required for running the algebraic simplification (blue line with xs on the left) and for deciding the satisfiability of the resulting simplified expression (Red line with dots on the right).

Note that the times in Figure 4 are in microseconds, as opposed to the seconds of Figure 3(a). The results show that the time required by the algebraic simplification is orders of magnitude smaller than the time required by the SMT solvers. Considering 64-bit words instead of 32-bit words does not change the results significantly.

5.5. Limitations

This approach is very attractive, due to its low complexity: we can reduce easily the analysis of a general MBA expression of high degree to the one of an expression of degree only 1. However, this process strongly depends on the generation process of the MBA.

We remark moreover that the degree of the polynomial f_d^{-1} is of primary interest, since the generation and the execution of the MBA expression strongly depend on this degree. On the contrary, the degree of f_d is used only in the generation of the MBA expression, and this generation depends only linearly on this degree. Following this remark, we can tweak the generation of the MBA expression, using polynomials from another family. These polynomials will not have the specific properties we used to reduce the MBA expression: in this case the reduction will fail.

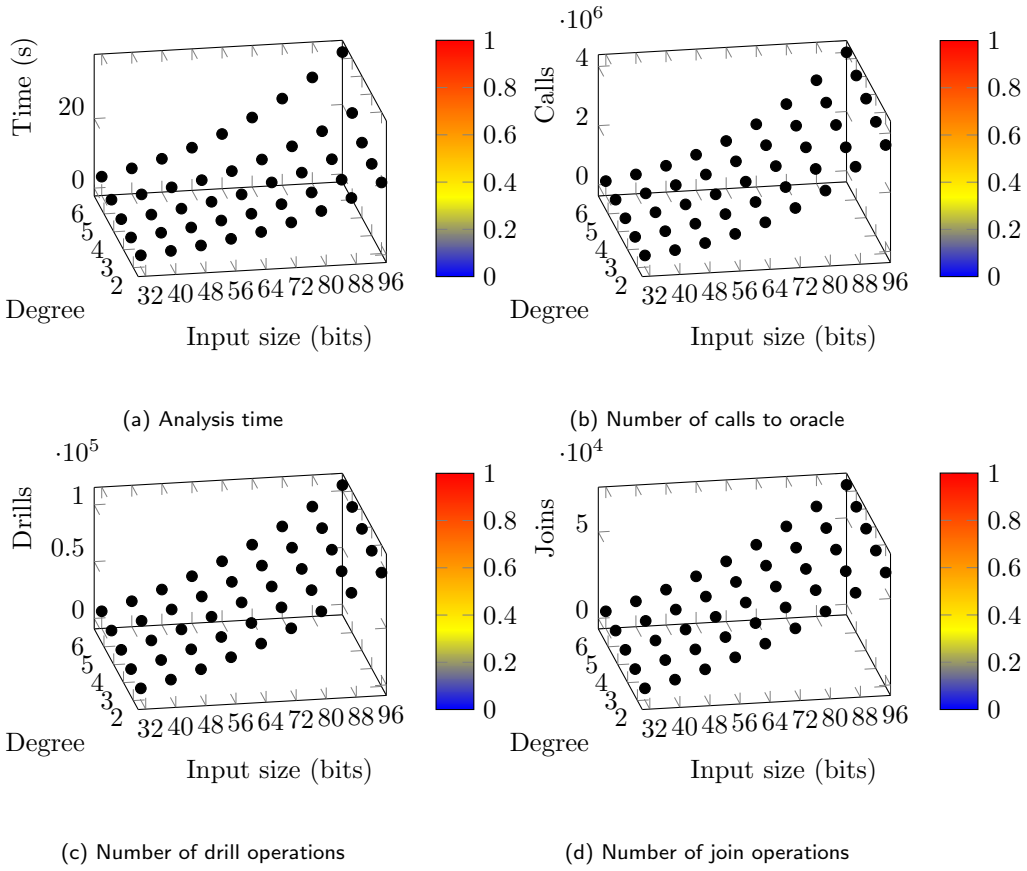


Figure 5. Application of the drill-and-join synthesis method to the synthesis of obfuscated linear MBA functions using polynomials of degree from 2 to 6 on an input size from 32 to 96 bits.

A more generic approach has then to be explored, to deal with more general MBA expressions. In Section 6 we abandon SMT-based methods and present our results on the application of the drill-and-join synthesis algorithm to this problem.

6. Synthesis-based Deobfuscation

In this section we use the drill-and-join synthesis method presented in Section 2.3 to directly reconstruct the unobfuscated function from the obfuscated conditional. The synthesis method considers the target obfuscated function as a black-box oracle, and reconstructs it by interrogating it and learning about its behavior.

While many symbolic execution techniques take advantage of concrete tests to improve efficiency, to the best of our knowledge dynamic synthesis has not

been investigated against obfuscated conditionals and never used to drive concolic execution.

We will show that the drill-and-join synthesis method is very effective, always outperforming the SMT solvers presented in Section 4 while solving a more complex problem (program synthesis instead of satisfiability). In fact, as the degree of the obfuscating polynomial increases, drill-and-join becomes able to synthesize the target function in less than the time required to obfuscate it in the first place. Finally, the method has the advantage of producing compact synthesized functions, which is particularly important if the function is to be used in some subsequent computation, as it is the case with concolic execution.

To get a sound synthesis of the target program, we have to explore the whole input space. In the case of our case study as presented in Section 2.2.2, the function we want to synthesize is a function on the 32-bit input variables x, x_1 and x_2, i.e. a 96-bit input space. Synthesized functions are validated against the black-box oracle by random testing. If the synthesis algorithm fails to synthesize a function or the random testing finds incongruences between the obfuscated and the synthesized functions, the attacker can either decide to consider both branches of the conditional as satisfiable, or decide to drop the branch to avoid adding suspicious branches to the control flow graph. In the first case the attacker risks of exploring unreachable branches of the graph, while in the second case the attacker risks not exploring reachable branches of the graph.

We have tested the drill-and-join method to synthesize the unobfuscated function within an obfuscated statement, using the obfuscated statement as a black box for the synthesis process. We tested the method on the comparison between an input and a constant described in Section 2.2.2, considering obfuscating polynomials of degree from 2 to 6 and adding spurious variables do obtain input sizes from 32 to 96 bits. The results are presented in Figure 5.

6.1. Results

Figure 5(a) presents the time necessary to synthesize the function. Comparing with the time required by the SMT solvers presented in Figure 3(a) it shows that the synthesis approach is faster than the SMT approach and scales better with the degree of the polynomial used in the obfuscation.

Figure 5(b) presents the number of calls to the oracle performed by the algorithm during the synthesis procedure. The number of calls does not significantly increase with the degree of the polynomials, confirming that the increase in computation time is mostly due to the increased cost of each call to the oracle. The computational time cost of the drill-and-join algorithm is exceeded by the time required by the oracle to answer the calls, showing the efficiency of the algorithm. We also note that we did not implement any of the optimizations to drill-and-join proposed by the author, like parallel implementation or subspace caching [16].

Finally, Figures 5(c) and 5(d) present the number of applications of the drill and join functions during the execution of the algorithm. They provide

additional insight on how the synthesis method scales with the input size and degree of the obfuscation polynomial.

We conclude that *black-box synthesis techniques are effective for reconstructing obfuscated conditionals.*

6.2. Limitations

The data show that the synthesis time increases exponentially with the input size, following a similar increase in the number of required calls to the oracle. This suggests that the obfuscator can effectively counteract this attack by increasing the number of variables used by the obfuscation polynomial, and consequently the input size. We will consider applying bitwise taint analysis to determine which bits of the input are in fact affecting the result and which are spurious additions, allowing the synthesis algorithm to focus only on the important bits.

6.3. Synthesis in Practice

In this section we briefly discuss how the synthesis approach would be used in a practical deobfuscation case and we provide some initial results on real-world obfuscated programs.

Most of the experiments conducted in our paper use KLEE and the MetaSMT framework, to be able to compare the efficiency of different SMT solvers against opaque conditionals. To apply such methods to real-world obfuscated programs, we have chosen instead to use the S2E symbolic execution engine [24]. This tool embeds the KLEE symbolic execution engine and allows us to directly analyze obfuscated binaries, thanks to QEMU's dynamic binary translation engine (DBT) frontends. S2E uses STP as its embedded SMT solver.

During execution of the target, we capture the queries to the STP SMT solver, and we analyze the time required to synthesize them using the drill-and-join algorithm.

For completeness, we also analyze the time required to solve the SMT formula with the Arybo tool [25] by Guinet et al. The method used by Arybo consists in identifying elementary symmetric functions, and specific patterns, and then apply rewriting rules to the computed ANF, through the interpretation of the MBA. The tool returns the ANF form, and is even able to invert the MBA in specific cases. This approach is particularly useful on constants, affine functions, point functions (that are false for all inputs except one), and relatively simple functions like CRC computations.

6.3.1. DRM Application

Following [25], we test synthesis on an obfuscated predicate obtained from a real life DRM obfuscated binary by Mougey and Gabriel [26] and built for Windows XP SP3.

The STP solver solves the predicate in 27 ms, but does not provide a synthetic representation of the predicate. By using the drill-and-join algorithm on the SMT formula sent to STP, in 25 ms we obtain:

```
(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
 NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
 NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
 (NTH 21 ARGS) (XOR T (NTH 22 ARGS)) (NTH 23 ARGS) (XOR T (NTH 24 ARGS))
 (XOR T (NTH 25 ARGS)) (XOR T (NTH 26 ARGS)) (NTH 27 ARGS) (NTH 28 ARGS))
```

which corresponds to the obfuscated operation: $x \oplus 01011100 = x \oplus 0x5C$.

This shows that that the drill-and-join algorithm has been able to get a concise representation of a predicate, which can be reinjected in the concolic analysis tool to simplify the rest of the concolic execution process. Finally, Arybo has been able to produce the function in this case in 150 ms, which is compatible with the 143 ms reported in [25].

6.3.2. Obfuscated Point Function

Some practical cases are currently not handled well by the drill-and-join algorithm. Consider the following point function also given in [25]:

```
uint64_t F(uint64_t X)
{
        uint64_t T = ((X+1)&(~X));
        uint64_t  C = ((T | 0x7AFAFA697AFAFA69) & 0x80A061440A061440)
              + ((~T & 0x10401050504) | 0x1010104);
        return C;
}
```

This function outputs the value `0xa061440b071544L` for any input, except for the input `0x7fffffffffffffffL` for which it outputs the value `0x80a061440b071544L`.

In this example, STP is able to find inputs to produce both outputs in just 3 ms, while the synthesis approach fails and after 5250 ms claims the predicate to be equivalent to the constant `0xa061440b071544L`. Arybo was able to synthesize the function is 31 ms, contrarily to the apparently pessimistic claim by the Arybo authors that estimate 36 years [25].

6.3.3. MBA Obfuscation

Finally, we tested the obfuscated predicate presented in Section 2.2.2 with MBA obfuscation of degree 4 on a 32-bit variable. We embedded the MBA-obfuscated conditional in a small C file, compiled it with GCC and examined the binary with S2E.

Compatibly with the results in the previous sections, STP decided satisfiability of the predicate in approximately 15 seconds, while drill-and-join synthesized it in approximately 1 second. However, Arybo was not able to produce a response in a reasonable time. This result is expected since the method implemented in Arybo is ineffective for the high-degree MBAs considered in our work. The authors report that 17 seconds are required for the analysis of an MBA of degree 2 consisting of two words of 9 bits. This suggests the worst for MBAs of higher degree and larger input sizes (not considered in [25]).

7. Related work

Some of the work presented in this paper has previously appeared as a poster [27] and as a technical report [28].

Recently Yadegari and Debray considered the problem of symbolically analyzing obfuscated conditionals [8]. They also focus on manipulating conditionals to hide them or to hide their relation with the inputs. They mainly consider the class of obfuscated conditionals that can be handled by taint-based analysis. We complement this approach by considering a class of obfuscation mechanisms that are inherently difficult to analyze with taint analysis. Yadegari et al. [9] also measured the impact of different taint analysis techniques on the quality of the deobfuscation for several off-the-shelf virtualization-based obfuscators.

Obfuscation. Many obfuscation techniques for hiding both data and control flow, along with evidence of their resilience against static analysis, can be found in the literature. The first formalization of the problem is due to Hada [29], while the first important theoretical result is Barak et al.'s proof of the impossibility of general-purpose virtual-black-box obfuscation [30]. On the other hand, Garg et al. proved the feasibility of general purpose indistinguishability obfuscators, where no such impossibility results exist [31]. However, while it has been proven to be resistant against algebraic attacks [32, 33], no practical implementation of such technique exists [34], thus this work focuses on the practically available and widely used MBA obfuscation instead.

Control Flow Obfuscation. The goal of the control flow flattening obfuscation method [35, 36] is to force an adversary to perform global analysis to understand local control flow transfers. Both forward and backward analyses are obstructed. However, control flow flattening protection mechanism can be reversed by applying suitable static optimization passes [37]. To thwart such attack methods, control flow flattening has to be enhanced by embedding a "difficult problem" in the compilation process to thwart static analyses such as constants or ranges propagation, etc. The SCFF protection scheme [10] is designed to obstruct flow-sensitive static analyses, which rely on accurate control flow information. This protection scheme is proved to be statically secure under the assumption that the initial value setting, which is done by obfuscated predicates concatenation, remains secret. This is an example of obfuscation mechanism whose robustness relies itself on an opaque conditional.

A similar approach is proposed by Wang et al. [38], adding a loop with an unsolved linear conjecture to hinder symbolic analysis.

Researchers have been trying to automatically counteract control flow flattening techniques for years [37], but not many effective tools are available, one notable exception being Johannes Kinder's Jakstab [39, 40, 41, 42]. However, at the current state of the art no automated approach is effective in reverse-engineering code obfuscated by state-of-the-art control flow flattening [43], severely crippling the capabilities of binary obfuscated binary reverse-engineering.

Opaque Predicates. Opaque predicates [44] have been introduced as a cheap control flow obfuscation technique, consisting of building predicates whose value is hard to determine statically. This adds unreachable paths to the control flow graph build by reverse-engineering the obfuscated code. Opaque predicates are *static* if they have the same value on each execution, and *dynamic* otherwise.

Static opaque predicates are *invariant* [17] if they always have the same truth value (i.e., correspond to true or false) and *contextual* [45] if their truth value depends on other variables (e.g., the comparison between a variable and an MBA-obfuscated constant in Section 2.2.2). We expect the synthesis approach proposed here to be effective against any kind of static predicate, as long as the predicate works a sufficiently small input space.

Dynamic opaque predicates [46, 47] are opaque predicates whose value can change between executions. Commonly family of predicates are correlated so that even if they change value, the execution trace remains the same. Hence, we expect dynamic synthesis to be effective on deobfuscating traces produced by dynamic opaque predicates, as long as the whole family of predicates is examined by the synthesis algorithm. If some of the predicates are outside the fragment of code to be synthesized, we expect synthesis to be misled and add unreachable traces to the control flow graph.

Detection of opaque predicates has been studied by Udupa et al. [37] and by Dalla Preda et al. [48], among others. We briefly discuss detection of MBA-obfuscated predicates in Section 3.2, but we only experiment with the time required to solve the predicates after they have been detected by such techniques or by a tool like LOOP [17].

White box cryptography. The goal of white box cryptography is to prevent an attacker from identifying and extracting the key in a block cipher encryption, even with a full control over the execution platform. To achieve this goal, white-box cryptography consists in implementing a specialized version of the algorithm that embeds the key k, and which is able to do only one of the two operations encrypt or decrypt [49]. This implementation is resilient in a white box context because it is difficult to extract the key k by observing the operations carried out by the program and because it is difficult to forge the decryption function starting from the implementation of the encryption function, and *vice versa*.

SMT solvers. Several applications of SMT solvers to software security can be found in static vulnerability checking, exploit generation and DRM evaluation. A survey of some practical applications is given in [50]. The LOOP tool [17] uses SMT solving to analyze the opaque predicates it finds. SMT-based input crafting for semi-automated cryptanalysis uses SMT solvers as equivalence checkers for verification of deobfuscation results of virtualization obfuscators. SMT solvers have been used for manually modeling licensing schemes.

8. Conclusions

We have investigated the effectiveness of dynamic synthesis when used to reconstruct obfuscated conditional statements. This is used by a concolic ex-

ecution engine to simplify the symbolic representation of the constraints over the variables, and to determine the satisfiability of the obfuscated conditionals examined.

We have considered MBA obfuscation as an example of a class of obfuscation techniques that are resistant to taint analysis and most static analysis methods used in code optimization.

We have started by evaluating the practical feasibility of MBA obfuscation when using polynomials of increasing degree. We have found that both the time required to produce the obfuscated statements and their size grow very rapidly, *de facto* preventing the use of polynomials of degree above 5 or 6.

We have evaluated the effectiveness of several SMT solvers in deciding the satisfiability of MBA obfuscated conditionals. We have observed a concolic execution engine like KLEE is significantly slowed down in determining the satisfiability of obfuscated conditionals, since the analysis time appears to increase exponentially with the degree of the polynomial used for the obfuscation.

We have presented an algebraic approach to simplify MBA obfuscation, reducing the complexity of the obfuscation with a polynomial of a given degree to the complexity of obfuscation with a polynomial of degree 1. The approach is able to determine whether the obfuscated function is a constant. This approach severely cripples MBA obfuscation, but strongly depends on the structure of the obfuscation and could be easily counteracted by slightly changing it, so a more general approach is required.

We have investigated the direct use of the drill-and-join dynamic black-box synthesis method, to simplify the representation of obfuscated conditionals. More generally this approach would be used by an attacker using concolic execution, along with an adapted strategy to drive the symbolic execution and employing synthesis as a more efficient rewriting strategy to boost the efficiency of an underlying SMT solver. We have found that drill-and-join can efficiently synthesize the obfuscated function, thus counteracting MBA obfuscation. Since dynamic synthesis does not depend on any property of MBA obfuscation, this result extends to other obfuscation methods.

This work proposes a synthesis-based attacker model. This complements other attacker models including those based on taint analysis, dynamic binary translation, optimization transformations, and so on. We conjecture that the synthesis approach will be particularly effective when combined with taint analysis, since taint analysis can be used to select the bits affecting the result of the obfuscated conditional and synthesis can produce the deobfuscated conditional as a function of those bits only.

Future work

We discuss some possible extensions of this work that we are exploring.

8.1. Attacker Model

In this work we try to define an attacker model representative of what can be realistically done by an adversary able to use both static and dynamic analysis

tools (symbolic and concolic execution engines) to reach his goal. Such a model can be used to assess the robustness of any obfuscation method.

As future work, we propose to study the integration of alternative black box approaches to drive the concolic execution of obfuscated programs. To the best of our knowledge, such an approach has never been investigated. The idea is to modify the usual concolic strategy to enable compact representations of obfuscated constraints, obtaining more synthetic and easy to read synthesized code. In addition, we expect the simplified constraints to be easier to solve or check for satisfiability.

8.2. Synthesis-supported SMT Solving

We conjecture that synthesis algorithm like drill-and-join could be used to simplify SMT formulae even in the general SMT solving scenario. For instance, a formula could be considered as a black-box oracle and synthesized as a preprocessing step, producing a compact synthesized formula that would then be subject to the normal satisfiability procedure. In this sense, synthesis would be used as a part of the simplification procedures already implemented by SMT solvers. Since synthesis seems to be sensitive mostly to the size of the input space and SMT solving mostly to the number of constructs, combining the strengths of the two techniques may result in a system more effective than synthesis or SMT solving alone.

8.3. Iterative Control Flow Graph Construction

When a conditional statement is found by the analyzer, it can be evaluated in a random point of the input space. If it evaluates to `true` (resp. `false`) then the `then` (resp. `else`) branch is reachable and can be explored by the analyzer, while the reachability of the `else` (resp. `then`) branch may require much more time to determine.

Consequently, we can quickly construct an underapproximation of the control flow graph by following only the branches we are certain about. Subsequently, the undecided branches can be re-examined to decide whether they lead to dead code or should be added to the graph.

The choice of the order in which to examine the branches is non-trivial. However, we note that the problem is similar to test generation for software model checking. Therefore, we expect that insight developed for fuzzying tools like SAGE [51] could be adapted to this aim.

References

[1] Quarkslab, Epona - code and data obfuscation to protect everywhere, `www.quarkslab.com`, 2016.

[2] P. Junod, J. Rinaldini, J. Wehrli, J. Michielin, Obfuscator-LLVM – software protection for the masses, in: B. Wyseur (Ed.), Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, IEEE, 2015, pp. 3–9. doi:`10.1109/SPRO.2015.10`.

[3] Oreans Technologies, Themida: Advanced windows software protection system, www.oreans.com/themida.php, 2013.

[4] Oreans Technologies, Code virtualizer: Total obfuscation against reverse engineering, www.oreans.com/codevirtualizer.php, 2014.

[5] VMProtect Software, VMProtect - new-generation software protection, www.vmprotect.ru, 2014.

[6] C. Collberg, S. Martin, J. Myers, J. Nagra, Distributed application tamper detection via continuous software updates, in: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12, ACM, New York, NY, USA, 2012, pp. 319–328. doi:10.1145/2420950.2420997.

[7] S. Udupa, S. Debray, M. Madou, Deobfuscation: reverse engineering obfuscated code, in: Reverse Engineering, 12th Working Conference on, 2005, pp. 10 pp.–. doi:10.1109/WCRE.2005.13.

[8] B. Yadegari, S. Debray, Symbolic execution of obfuscated code, in: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, ACM, New York, NY, USA, 2015, pp. 732–744. doi:10.1145/2810103.2813663.

[9] B. Yadegari, B. Johannesmeyer, B. Whitely, S. Debray, A generic approach to automatic deobfuscation of executable code, in: Security and Privacy (SP), 2015 IEEE Symposium on, 2015, pp. 674–691. doi:10.1109/SP.2015.47.

[10] J. Cappaert, B. Preneel, A general model for hiding control flow, in: Proceedings of the 10th ACM workshop on Digital Rights Management (DRM 2010), 2010, pp. 35–42.

[11] A. Biryukov, A. Shamir, Structural cryptanalysis of SASAS, Journal of Cryptology 23 (2010) 505–518.

[12] O. Billet, H. Gilbert, C. Ech-Chatbi, Selected Areas in Cryptography: 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 227–240. doi:10.1007/978-3-540-30564-4_16.

[13] S. Drape, Intellectual Property Protection using Obfuscation, Technical Report RR-10-02, 2010.

[14] Irdeto, Cloaked CA solution, www.irdeto.com, 2014.

[15] Y. Zhou, A. Main, Y. X. Gu, H. Johnson, Information hiding in software with mixed boolean-arithmetic transforms., in: S. Kim, M. Yung, H.-W. Lee (Eds.), WISA, volume 4867 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 61–75.

[16] R. Balaniuk, Drill and join: A method for exact inductive program synthesis, in: M. Proietti, H. Seki (Eds.), Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers, volume 8981 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 219–237. doi:10.1007/978-3-319-17822-6_13.

[17] J. Ming, D. Xu, L. Wang, D. Wu, LOOP: logic-oriented opaque predicate detection in obfuscated binary code, in: I. Ray, N. Li, C. Kruegel (Eds.), Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015, ACM, 2015, pp. 757–768.

[18] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, 2004.

[19] C. Cadar, D. Dunbar, D. Engler, KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 209–224.

[20] F. Haedicke, S. Frehse, G. Fey, D. Große, R. Drechsler, metaSMT: Focus on your application not on solver integration, in: M. K. Ganai, A. Biere (Eds.), Proceedings of the First International Workshop on Design and Implementation of Formal Tools and Systems, Austin, USA, November 3, 2011, volume 832 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2011.

[21] V. Ganesh, D. L. Dill, A decision procedure for bit-vectors and arrays, in: Proceedings of the 19th International Conference on Computer Aided Verification, CAV'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 519–531.

[22] L. De Moura, 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, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.

[23] R. Brummayer, A. Biere, Boolector: An efficient SMT solver for bit-vectors and arrays, in: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 174–177. doi:10.1007/978-3-642-00768-2_16.

[24] V. Chipounov, V. Kuznetsov, G. Candea, S2e: a platform for in-vivo multi-path analysis of software systems., in: R. Gupta, T. C. Mowry (Eds.),

ASPLOS, ACM, 2011, pp. 265–278. URL: http://dblp.uni-trier.de/db/conf/asplos/asplos2011.html#ChipounovKC11.

[25] A. Guinet, N. Eyrolles, M. Videau, Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions, in: GreHack 2016, 2016.

[26] C. Mougey, F. Gabriel, Drm obfuscation versus auxiliary attacks, in: Recon, 2014.

[27] F. Biondi, S. Josse, A. Legay, Comparative evaluation of the effectiveness of constraint solvers against opaque conditionals, in: Security and Privacy (SP), 2015 IEEE Symposium on, 2015. Poster session.

[28] F. Biondi, S. Josse, A. Legay, T. Sirvent, Effectiveness of Synthesis in Concolic Deobfuscation, 2015. URL: https://hal.inria.fr/hal-01241356, working paper or preprint.

[29] S. Hada, Zero-knowledge and code obfuscation, in: T. Okamoto (Ed.), Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings, volume 1976 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 443–457. doi:10.1007/3-540-44448-3_34.

[30] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, K. Yang, On the (im)possibility of obfuscating programs, Electronic Colloquium on Computational Complexity (ECCC) 8 (2001).

[31] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, B. Waters, Candidate indistinguishability obfuscation and functional encryption for all circuits, in: 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA, IEEE Computer Society, 2013, pp. 40–49. doi:10.1109/FOCS.2013.13.

[32] Z. Brakerski, G. N. Rothblum, Virtual black-box obfuscation for all circuits via generic graded encoding, in: Y. Lindell (Ed.), Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings, volume 8349 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 1–25. doi:10.1007/978-3-642-54242-8_1.

[33] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, A. Sahai, Protecting obfuscation against algebraic attacks, in: P. Q. Nguyen, E. Oswald (Eds.), Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings, volume 8441 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 221–238. doi:10.1007/978-3-642-55220-5_13.

[34] S. Banescu, M. Ochoa, N. Kunze, A. Pretschner, Idea: Benchmarking indistinguishability obfuscation - a candidate implementation, in: F. Piessens, J. Caballero, N. Bielova (Eds.), Engineering Secure Software and Systems - 7th International Symposium, ESSoS 2015, Milan, Italy, March 4-6, 2015. Proceedings, volume 8978 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 149–156.

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

[36] C. Wang, A security architecture for survivability mechanisms, Ph.D. thesis, University of Virginia, 2001.

[37] S. K. Udupa, S. K. Debray, M. Madou, Deobfuscation: Reverse engineering obfuscated code, in: In WCRE 05: Proceedings of the 12th Working Conference on Reverse Engineering, IEEE Computer Society, 2005, pp. 45–54.

[38] Z. Wang, J. Ming, C. Jia, D. Gao, Linear obfuscation to combat symbolic execution, in: V. Atluri, C. Díaz (Eds.), Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings, volume 6879 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 210–226. doi:10.1007/978-3-642-23822-2_12.

[39] J. Kinder, Static analysis of x86 executables (Statische Analyse von Programmen in x86-Maschinensprache), Ph.D. thesis, Darmstadt University of Technology, 2010. URL: http://tuprints.ulb.tu-darmstadt.de/2338/.

[40] J. Kinder, Towards static analysis of virtualization-obfuscated binaries, in: 19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012, IEEE Computer Society, 2012, pp. 61–70. URL: http://dx.doi.org/10.1109/WCRE.2012.16. doi:10.1109/WCRE.2012.16.

[41] J. Kinder, D. Kravchenko, Alternating control flow reconstruction, in: V. Kuncak, A. Rybalchenko (Eds.), Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings, volume 7148 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 267–282. URL: http://dx.doi.org/10.1007/978-3-642-27940-9_18. doi:10.1007/978-3-642-27940-9_18.

[42] J. Kinder, H. Veith, Jakstab: A static analysis platform for binaries, in: A. Gupta, S. Malik (Eds.), Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings, volume 5123 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 423–427. URL: http://dx.doi.org/10.1007/978-3-540-70545-1_40. doi:10.1007/978-3-540-70545-1_40.

[43] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, E. R. Weippl, Protecting software through obfuscation: Can it keep pace with progress in code analysis?, ACM Comput. Surv. 49 (2016) 4.

[44] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, 1997.

[45] B. Coppens, B. D. Sutter, J. Maebe, Feedback-driven binary code diversification, TACO 9 (2013) 24.

[46] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, Y. Zhang, Experience with software watermarking, in: 16th Annual Computer Security Applications Conference (ACSAC 2000), 11-15 December 2000, New Orleans, Louisiana, USA, IEEE Computer Society, 2000, pp. 308–316.

[47] D. Xu, J. Ming, D. Wu, Generalized dynamic opaque predicates: A new control flow obfuscation method, in: Information Security - 19th International Conference, ISC 2016, Honolulu, Hawaii, USA, September 7-9, 2016, Proceedings, 2016.

[48] M. D. Preda, M. Madou, K. D. Bosschere, R. Giacobazzi, Opaque predicates detection by abstract interpretation, in: M. Johnson, V. Vene (Eds.), Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings, volume 4019 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 81–95. doi:10.1007/11784180_9.

[49] S. Chow, P. Eisen, H. Johnson, P. C. V. Oorschot, A white-box DES implementation for DRM applications, in: In Proceedings of ACM CCS-9 Workshop DRM, Springer, 2002, pp. 1–15.

[50] J. Vanegue, S. Heelan, SMT solvers in software security, in: E. Bursztein, T. Dullien (Eds.), 6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings, USENIX Association, 2012, pp. 85–96.

[51] P. Godefroid, M. Y. Levin, D. Molnar, SAGE: Whitebox fuzzing for security testing, Commun. ACM 55 (2012) 40–44.