Effectiveness of Synthesis in Concolic Deobfuscation

Fabrizio Biondi, Sébastien Josse, Axel Legay, Thomas Sirvent

Effectiveness of Synthesis in Concolic Deobfuscation

Fabrizio Biondi*,Sébastien Josse†,Axel Legay*, Thomas Sirvent†
*Inria
Email: {fabrizio.biondi,axel.legay}@inria.fr
†DGA (French Ministry of Defense)
Email: sebastien.josse@polytechnique.edu, thomas.sirvent@m4x.org

***Abstract*—The obfuscation of conditional statements is a simple and efficient way to disturb the identification of the control flow graph of a program. Mixed Boolean arithmetics (MBA) techniques provide concrete ways to achieve this obfuscation of conditional statements. In this work, we study the effectiveness of automated deobfuscation of MBA obfuscation, using algebraic, SMT-based and synthesis-based techniques. We first experimentally ascertain the practical feasibility of MBA obfuscation, and we show how its complexity can be greatly reduced by algebraic simplification. We study using SMT-based approaches with different state-of-the-art SMT solvers to counteract MBA obfuscation. We also consider synthesis-based deobfuscation and find it to be more effective than SMT-based deobfuscation. We discuss complexity and limits of all methods, and conclude that MBA obfuscation is not effective enough to be considered a reliable method for control flow or white-box obfuscation.**

1. Introduction

To protect a system for malware infection it is necessary to be able to analyze the behavior of an executable binary file and classify it as dangerous. The behavior of the binary is modeled by its control flow graph, thus the extraction of the control flow graph is a fundamental step in malware fingerprinting and classification. *Control flow obfuscation* transformations are designed to hamper the reverse engineering of the control flow graph of the system. As a consequence of control flow obfuscation, the analyst is unable to extract manually or statically a relevant representation of the program, starting with the control flow graph. It is therefore necessary to execute (concretely or symbolically) the program to determine the values of these expressions and thus reconstruct the control flow graph.

The branching nodes of the control flow graph are its conditional statements. Standard symbolic and concolic (concrete/symbolic) analyses consider conditional statements as a set of constraints and then use satisfiability modulo theory (SMT) constraint solvers to decide the satisfiability of a conditional expression, as depicted in Figure **??**. The cost of reverse engineering can be greatly increased by adding complex unreachable parts in the code and hiding their unreachability by *obfuscating the conditionals* leading to such parts [**?**]. Obfuscation of conditional statements aims to make it difficult or impossible for an analyst to determine the truth value or even satisfiability of conditional statements.

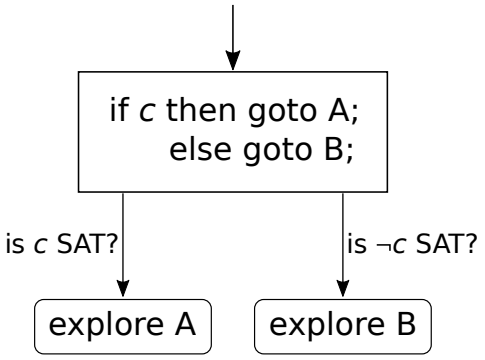


Figure 1. Concolic execution. When a conditional statement is found the guard c and its negation are tested for satisfiability, and the two branches are explored accordingly.

We propose the replacement of SMT solvers with dynamic black-box synthesis techniques in this reverse engineering scenario. We show that *synthesis techniques are more effective than SMT solvers in determining satisfiability of obfuscated conditionals*.

Fundamental research on cryptographic obfuscation is currently very active, thanks to recent breakthroughs. This theoretical approach remains however far from being applicable in practice [**?**]. For this reason, in this paper we focus on software-only techniques aiming at increasing the cost of reverse engineering. More precisely, we consider the obfuscation method based on mixed Boolean arithmetics (MBA) proposed by Zhou et al. [**?**], [**?**]. Informally, a polynomial is used as a "shell" to protect an expression together with some linear MBA identities, obtaining a multivariate polynomial from which it is difficult to find back the original expression. This obfuscation mechanism is currently used in specialized compilation chains [**?**]. MBA obfuscation is representative of the principle of obfuscation by embedding a difficult (assumed NP-complete) problem into an implementation. We also consider whether MBA obfuscation could be effec-

tively used as white-box cryptography, i.e. to hide a secret constant (e.g. a private key) inside the code of a program by obfuscating it.

We thoroughly evaluate the effectiveness and practical applicability of MBA obfuscation. We evaluate the cost of MBA obfuscation in terms of compilation time and increase in execution time of the obfuscated program, obtaining upper bounds on the degree of the main polynomial that can be employed for the obfuscation technique in practice.

We present an algebraic simplification technique that exploits the particular structure of MBA obfuscation polynomials in [**?**] to reduce the degree of the polynomial used for the obfuscation to 1, consequently reducing the required analysis time for any deobfuscation approach.

We assess the security of MBA obfuscation against symbolic execution based on SMT solvers and concolic execution based on an especially crafted synthesis method. We show that these techniques are more effective for deobfuscation than the attacks based on computer algebra considered by Zhou et al [**?**].

More precisely, we compare the effectiveness of several state-of-the-art SMT solvers in determining the satisfiability of MBA-obfuscated conditionals. We show that the time required for SMT solvers to deobfuscate the conditionals increases exponentially with the degree of the polynomial used in the obfuscation. This is justified by the exponential increase in the number of constructs of the obfuscated conditional statements.

We apply then a dynamic synthesis method targeted at the MBA obfuscation technique and show its effectiveness compared to the approach based on SMT solving. Our results suggest that analogous synthesis approaches could be implemented in SMT solvers to greatly improve their effectiveness in solving complicated black-box statements.

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.
2) We test the effectiveness of SMT-based techniques for determining the truth values of MBA-obfuscated conditional statements.
3) We present an algebraic simplification approach that reduces the complexity of polynomial MBA deobfuscation to the deobfuscation of linear MBA obfuscation.
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.

All experiments are conducted on a Intel Core i5 1.60-2.30 GHz.

Outline. The remainder of this paper is structured as follows: Section **??** introduces MBA obfuscation, the drill-and-join synthesis method, and the mathematical theory necessary to understand them. Section **??** discusses the practical feasibility of MBA obfuscation, determining bounds on the degree of the polynomials that the technique can use in practice. Section **??** tests the effectiveness of SMT solvers in determining the truth value of obfuscated conditionals and gives an algebraic simplification technique to greatly reduce the deobfuscation time. Section **??** evaluates the drill-and-join synthesis method on the same problem. Finally, Section **??** concludes the work and points out possible future research directions.

Related work

Recently Yadegari and Debray also considered the problem of executing obfuscated code and conditionals [**?**]. However, the obfuscation techniques they consider are simpler than MBA obfuscation, so adapting the taint analysis deobfuscation they propose to our case would not be straightforward.

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 [**?**], while the first important theoretical result is Barak et al.'s proof of the impossibility of general-purpose virtual-black-box obfuscation [**?**]. On the other hand, Garg et al. proved the feasibility of general purpose indistinguishability obfuscators, where no such impossibility results exist [**?**]. However, while it has been proven to be resistant against algebraic attacks [**?**], [**?**], no practical implementation of such technique exists [**?**], thus this work focuses on the practically available and widely used MBA obfuscation instead.

Control Flow Obfuscation. The goal of the Control Flow Flattening (CFF) obfuscation [**?**] method is to force an adversary to perform global analysis to understand local control flow transfers. Both forward and backward analyses are obstructed. However, the CFF protection mechanism can be inversed by applying suitable static optimization passes [**?**]. To thwart such attack methods, the CFF mechanism 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 [**?**] 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 obfuscated constant.

White box cryptography. Another area where obfuscated constant may be used is cryptographic algorithms. The goal 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 that 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 [**?**]. 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*. We remark that this approach is specifically designed for specific block cipher algorithms, and cannot be used in a more general context.

Deobfuscation methods.

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 [**?**]. 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.

Program synthesis. The program synthesis technique described in [**?**] has been applied to Conficker [**?**] and MyDoom malware deobfuscation. Program synthesis techniques for learning programs from traces [**?**], taking advantage of the target program's dynamic states observation, can be applied to binary reverse engineering.

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 ring $\mathbb{Z}/2^n\mathbb{Z}$ and by $\mathbb{F}_2^n$ the ring $(\mathbb{Z}/2\mathbb{Z})^n$ of $n-$tuples of elements in $(\mathbb{Z}/2\mathbb{Z})$.

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. [**?**] 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. [**?**] 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. 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 [**?**] 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. 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

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

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

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

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 **??** 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. [**?**]. While it can be used to obfuscate software to hide secret constants, intermediate values, and algorithms from reverse engineering, we will focus in this work only on obfuscating constants.

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 **??**. 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. We want to construct a polynomial in m variables $x_1, \ldots, x_m$ that is equivalent to k.

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 [**?**]. 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. [**?**, 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 [**?**, Appendix A]. We will obfuscate this conditional statement with the MBA obfuscation method described above with different polynomial degrees and evaluate the effectiveness of different analysis methods on it.

Assume that the conditional to be obfuscated compares some input IN with the constant $k = 0\text{x}87654321$:

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

Using 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:

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

The following obfuscated conditional is generated:

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

f = 2284837645 + 272908530*a + 136454265*b
     + 409362795*x + 135832444*d
     + 4159134852*e + 415760384*a*a
     + 415760384*a*b + 1247281152*a*x
     + 2816475136*a*d + 1478492160*a*e
     + 3325165568*b*b + 2771124224*b*x
     + 1408237568*b*d + 2886729728*b*e
     + 4156686336*x*x + 4224712704*x*d
     + 70254592*x*e + 1428160512*d*d
     + 1438646272*d*e + 1428160512*e*e;
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:

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

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 **??**. However, the SMT-based and synthesis-based deobfuscation methods presented in Sections **??** and **??** 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 [**?**].

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 $(v_i)_{1 \leq i \leq m}$, if $\exists x_0 \in \mathbb{F}_2^n$ and $\exists v_0$ such that $f(x_0) = 1$ and $v_0(x_0) = 1$, 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

$$f(x) \quad = \quad \gamma f(x) \oplus v_0(x_0) \quad .$$

The required x_0 such that $f(x_0) = 1$ 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 :

$$\begin{aligned} f(x) &= \gamma^1 f(x) \oplus v_0(x_0) \\ &= \gamma^2 f(x) \oplus v_1(x_1) \oplus v_0(x_0) \\ &= \gamma^m f(x|y) \oplus \bigoplus_{i=0}^{m-1} v_i(x_i) \\ &= \bigoplus_{i=0}^{m-1} v_i(x_i) \quad . \end{aligned}$$

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

In Section **??** 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 [**?**]. Consider the target function $f(x|y) = y \vee (\neg x \wedge \neg y)$. We apply the drill function with $x_0 = y_0 = F$ and obtain two subspace problems $f(x_0|y)$ and $f(x|y_0)$. Since $\delta^1 f = \mathbb{F}(f(X|Y), X_0, Y_0)$ is not the zero map we apply drill recursively with $x_1 = y_1 = T$, obtaining the subspace problems $\delta^1 f(x_1|y)$ and $\delta^1 f(x|y_1)$. Since $\delta^2 f = \mathbb{F}(\delta^1 f(X|Y), X_1, Y_1) = F$ is the zero map, the recursion terminates. Now we use the join function to solve the four obtained one-dimensional subspace problems $f(x_0|y)$, $f(x|y_0)$, $\delta^1 f(x_1|y)$, and $\delta^1 f(x|y_1)$ using basis $v_0(x) = x, v_1(x) = T$. Consider the problem $f(x|y_0)$. We apply the join function two times with bases $x_0 = F, v_0(x) = x$ and $x_1 = T, v_1(x) = T$ respectively to obtain the zero map, so we reconstruct it as $f(x|y_0) = v_0(x) \oplus v_1(x) = x \oplus T$. Similarly we obtain $f(x_0|y) = T$, $\delta^1 f(x_1|y) = y$, and $\delta^1 f(x|y_1) = x$. Finally, we reconstruct the target function as $f(x|y) = (f(x_0|y) \wedge f(x|y_0)) \oplus (\delta^1 f(x_1|y) \wedge \delta^1 f(x|y_1)) = (T \wedge x \oplus T) \oplus (y \wedge x) = x \oplus (x \wedge y)$ which is equivalent to 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 **??**. In this section we evaluate the amount of time required to generate obfuscated code and its size. The results are depicted in Figure **??**.

The blue line with xs on the left in Figure **??** represents the amount of time required by the obfuscation system to produce an obfuscated conditional equivalent to the comparison of an input and a constant presented in Section **??**. As the graph shows, the time required to produce the obfuscated statement grows exponentially with the degree of the polynomial used for the obfuscation. The time required to obfuscate code using the MBA obfuscation grows quickly enough to make it impractical for large degrees.

We note that the author of the code needs to obfuscate it only once, so the increase in compilation time does not affect the development process, and the additional time cost due to obfuscation can be paid only once, i.e. before releasing the final product. However, the attacker also needs to deobfuscate the code only once to obtain unobfuscated source code. We will return on this when we discuss deobfuscation time in Section **??**.

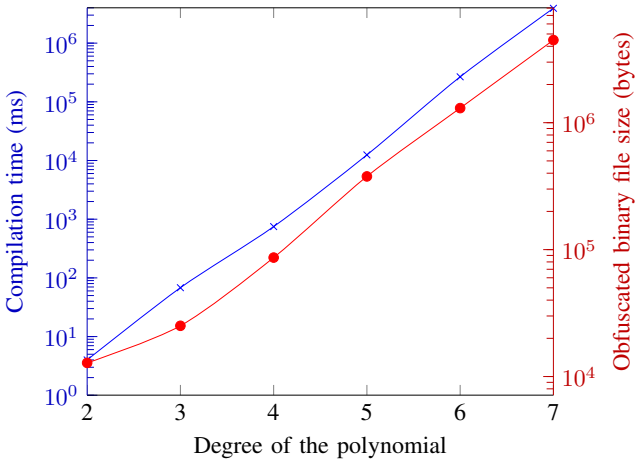


Figure 2. Cost of MBA obfuscation of the constant comparison case study in Section **??** using obfuscation polynomials of different degrees. Blue line with xs on the left: compilation time in milliseconds. Red line with dots on the right: obfuscated binary size in bytes.

The red line with dots on the right in Figure **??** represents the size of the obfuscated binary file corresponding to the obfuscated constant comparison statement from Section **??**. 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 $\sim$ 370 kB, making the total size of the binary program impractically large.

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

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 **??**. 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.

Once the attacker has identified a suspicious multivariate polynomial in the code, he can evaluate it with different values of the variables $x_1, \ldots, x_t$. If it is the obfuscated expression of a constant it will always evaluate to the constant, while if it is the obfuscated expression of a linear function it will be easy to synthesize the function itself. We develop the latter point in more details in Section **??**.

4. SMT Solver-based Deobfuscation

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

4.1. Experimental Setup

We use the LLVM [**?**] 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 [**?**] and the multi-solver support in KLEE provided by the MetaSMT framework [**?**].

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.

4.2. Solvers

We use the STP, Z3 and Boolector SMT solvers.

STP git-12/02/2015. The STP solver [**?**] 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 [**?**] 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. Z3 is the only SMT solver able to produce a counterexample to unsatisfiable statements in the logic we use; while we did not use this functionality in our experiments, it would be very useful in a more generic concolic execution environment.

Boolector 1.5.118. Boolector [**?**] 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 **??** 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.

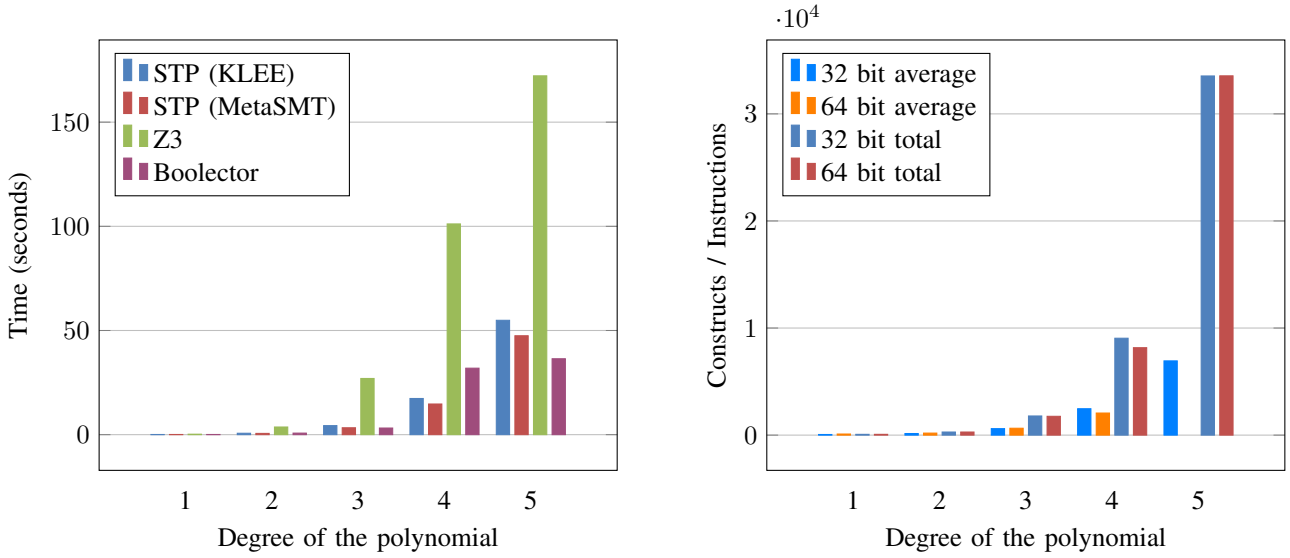
The data shows that Z3 is the slowest SMT solver for handling obfuscated conditionals, and that Boolector is slightly faster than STP on higher degree cases. The two different implementations of STP, integrated in KLEE and via metaSMT, do not differ significantly in solution time, showing that metaSMT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

The graph in Figure **??** shows that the MBA obfuscation method is effective in slowing down the analysis of conditional statements even with a small degree of the main polynomial used. We conclude that *the MBA obfuscation method is an acceptable method for slowing down control flow deobfuscation based on SMT techniques*, i.e. when a large number of conditional statements would have to be deobfuscated and the attacker would not be able to spend the required hundreds of seconds for each one.

On the other hand, an attacker interested in retrieving a particular constant from the system could use a concolic execution approach to execute the code of the system until the desired point is reached and then employ SMT solvers only on one given conditional to retrieve an important obfuscated constant, e.g. a private key. In this last case the attacker could spend days deobfuscating the expression of the constant with an SMT solver, so the level of obfuscation proposed would be insufficient against such attacks. We conclude that *the MBA obfuscation method is not effective enough to be used as white-box obfuscation against SMT-based techniques* for the purpose of hiding e.g. secret keys in code and similar critical data.

Figure **??** gives the average number of constructs per (solver) query and the total 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 conclude 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



(a) Analysis time

(b) Average constructs per query and total instructions

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

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, while SMT solvers are able to break MBA obfuscation given enough time, they are not efficient enough to consider them a sufficient measure of deobfuscation for the concolic execution scenario. In Section **??** 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 **??** 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.

4.4. 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 [**?**]), such a construction can be recognized, analyzed and simplified algebraically, thank to the specific form of the MBA-obfuscated conditional, if they contain a constant. 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 constant.

Note that in general distinguishing point functions from constant functions is very hard, usually requiring a full exploration of the input space. This approach allows us to efficiently discriminate between obfuscated constant and point functions. If we conclude that the function is a constant, a simple evaluation sufficient to know its satisfiability. If the function is not a constant, then both branches of the conditional are satisfiable.

This simplification provides a way to *consider only expressions of degree* 1*, from expressions of arbitrary degree, with a computation cost similar to the reading of the original expression*. This result requires that the expression has been built exactly as described in [**?**], since we have in particular used the specific form of the polynomials.

The full details of the algebraic simplification are given in the Appendix due to space constraints. Here we provide a sketch of the technique and an illustrative example, then we discuss complexity, experimental results and limitations of the technique.

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 constant $f_d(k)$. Finally, we compute the value k hidden in the expression as $f_d^{-1}(f_d(k))$.

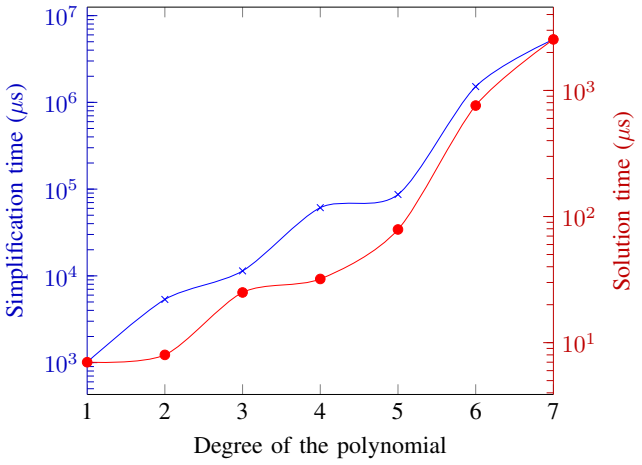


Figure 4. Algebraic simplification time for obfuscation polynomials different degrees. Blue line with xs on the left: Simplification time in microseconds. Red line with dots on the right: Solution time in microseconds.

4.4.1. Example. We use the example given in **??**. 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: $b_0 + b_1\, c + b_2\, c^2 = 2271560481 = 0\text{x}87654321$.

4.4.2. 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 constant 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.

4.4.3. Results. The time required for the algebraic simplification is presented in Figure **??**. 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 **??** are in microseconds, as opposed to the milliseconds of Figure **??** and the seconds of Figure **??**. The results show that the time required by the algebraic simplification is orders of magnitude smaller than the time required by the SMT solvers and by the obfuscation process itself. Considering 64-bit words instead of 32-bit words does not change the results significantly.

4.4.4. 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. The algebraic reduction fails if the obfuscated statement is not a constant. However, knowing that the statement is not a constant is sufficient to determine that both it and its negation are satisfiable.

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.

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

5. Synthesis-based Deobfuscation

In this section we use the drill-and-join synthesis method presented in Section **??** 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 the drill-and-join method 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 **??** while solving a more complex

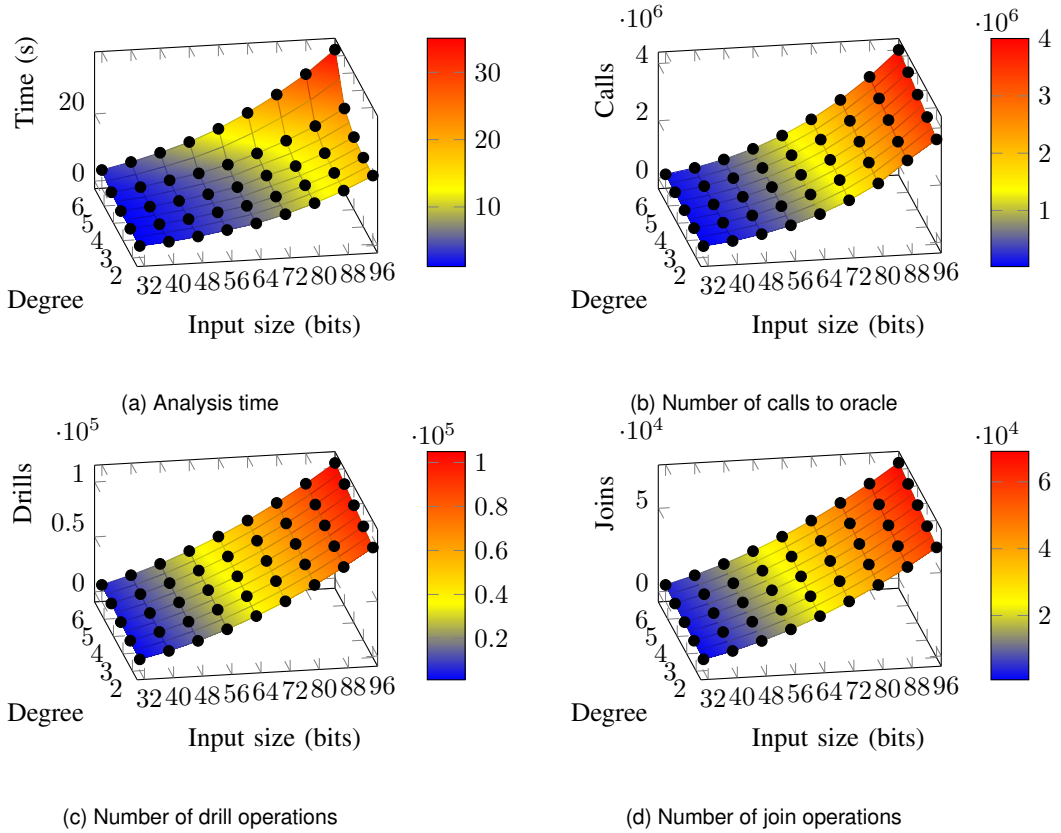


Figure 5. Application of the drill-and-join synthesis method to the synthesis of obfuscated linear MBA functions using polynomials of degree from 2 to 6 on an input size from 32 to 96 bits.

problem (program synthesis instead of satisfiability). In fact, as the degree of the obfuscating 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 **??**, 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 have considered obfuscating polynomials of degree from 2 to 6 and input sizes from 32 to 96 bits. The results are presented in Figure **??**.

Figure **??** presents the time necessary to synthesize the function. Comparing with the time required by the SMT solvers presented in Figure **??** 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. In fact, comparison with the compilation cost required to construct the obfuscated polynomial presented in Figure **??** shows that the synthesis procedure takes approximately the same time as the obfuscation for polynomials of degree 6, and since it grows more slowly with the degree of the polynomial we can expect synthesis to be faster than obfuscation for polynomials of higher degree.

The computation time seems to grow with the degree of the polynomial similiarly to the execution time shown as the red line with dots in Figure **??**. This suggests that the number of calls to the oracle does not significantly increase with the degree of the obfuscation polynomial, while the cost of each call does.

Figure **??** 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 [**?**].

Finally, Figures **??** and **??** 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 *employing synthesis techniques to reconstruct the obfuscated function by interrogating it as a black box is an effective countermeasure to MBA obfuscation*. In fact, it proves the existence of practical approaches to circumvent MBA obfuscation, suggesting that MBA obfuscation should not be considered as an effective way to protect industrial secrets such as private keys and algorithms. We expect drill-and-join synthesis to be less effective in synthesizing non-linear functions, and we will consider additional investigation in the capabilities and limits of the method as future work.

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 dealing with such large input sizes as future work, possibly combining it with a generalization of the simplification technique presented in Section **??**.

6. Conclusions

We have investigated in this paper the robustness of a generic obfuscation mechanism based on mixed Boolean arithmetics (MBA), which can be used to hide a function or constant value in the code of a program, possibly creating obfuscated conditional statements.

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 that even though the analysis is slowed down, a symbolic execution engine like KLEE is sufficient to determine the satisfiability of obfuscated conditionals. While the analysis time appears to increase exponentially with the degree of the polynomial used for the obfuscation, the fact that only low-degree polynomials can be used in practice means that the analysis will still terminate successfully in a reasonable time.

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.

It should be noticed that we are not yet proposing a unified attacker model. We hope that our work, mainly centered on experimentations, provides a first stage towards the definition of a realistic attacker model.

Future work

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

6.1. 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.

We plan to further explore this approach and fully automate it as a tool for deobfuscation of compiled code. Such a tool will be used as a benchmark to evaluate the effectiveness of obfuscation methods. As a side effect, we also expect to provide bases for more effective code analysis tools, even when the target is not obfuscated, by using synthesis to improve existing rewriting techniques.

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. We conjecture that such an approach may be applied to a large class of obfuscated constraints, and not only to the ones produced by MBA obfuscation.

Some of the attacks presented in this work can be combined to increase their effectiveness. In particular, the algebraic approach in Section **??** could be used to reduce the degree of the obfuscation polynomial to 1, thus simplifying the problem to be solved by the SMT-based or synthesis-based approach. In particular, this could be successful in

enabling the synthesis approach to handle cases with a large input size.

6.2. Evaluation of More Complex Obfuscation Cases

In this paper, we have limited our investigations to simple obfuscated conditional statements, namely constant comparisons and linear functions. The MBA obfuscation method can be used to hamper both data and control flow analysis, making an algorithm difficult to understand. It can also be used to hide a more complex formula than a linear function, resulting in the embedding of a more difficult problem. However, our MBA obfuscation cost analysis suggests that such advanced methods may be impractical.

We conjecture that an obfuscated constraint may be as difficult to break as the embedded problem, independently of the mechanism used to diversify and hide its representation, against an attacker using symbolic or concolic execution.

6.3. 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.

The greatest challenge to this approach is determining the soundness of the synthesis procedure. Validating a synthesized program against its black-box oracle is a notoriously complex problem, and often depend on the intervention of a user [**?**]. For the drill-and-join approach, our experiments provide evidence that the algorithm could be sound when synthesizing affine functions. In this case, an affinity test could be developed and used on the black-box to predict the reliability of the synthesized program in emulating the original one.

6.4. 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 [**?**] could be adapted to this aim.

References

[1] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

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

[3] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms." in *WISA*, ser. Lecture Notes in Computer Science, S. Kim, M. Yung, and H.-W. Lee, Eds., vol. 4867. Springer, 2007, pp. 61–75.

[4] H. Johnson, Y. Gu, and Y. Zhou, "System and method for interlocking to protect software-mediated program and device behaviours," Aug. 28 2008, wO Patent App. PCT/CA2008/000,333.

[5] C. Liem, Y. X. Gu, and H. Johnson, "A compiler-based infrastructure for software-protection," in *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '08. New York, NY, USA: ACM, 2008, pp. 33–44.

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

[7] S. Hada, "Zero-knowledge and code obfuscation," in *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*, ser. Lecture Notes in Computer Science, T. Okamoto, Ed., vol. 1976. Springer, 2000, pp. 443–457.

[8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 8, no. 057, 2001.

[9] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and 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.

[10] Z. Brakerski and G. N. Rothblum, "Virtual black-box obfuscation for all circuits via generic graded encoding," in *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, ser. Lecture Notes in Computer Science, Y. Lindell, Ed., vol. 8349. Springer, 2014, pp. 1–25.

[11] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai, "Protecting obfuscation against algebraic attacks," in *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, ser. Lecture Notes in Computer Science, P. Q. Nguyen and E. Oswald, Eds., vol. 8441. Springer, 2014, pp. 221–238.

[12] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 735–746, 2002.

[13] S. K. Udupa, S. K. Debray, and 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.

[14] J. Cappaert and 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.

[15] S. Chow, P. Eisen, H. Johnson, and 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.

[16] "SMT solvers in software security," in *Presented as part of the 6th USENIX Workshop on Offensive Technologies*. Berkeley, CA: USENIX, 2012.

[17] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, May 2010, pp. 215–224.

[18] P. Porras, H. Saïdi, and V. Yegneswaran, "A foray into conficker's logic and rendezvous points," in *Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, ser. LEET'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 7–7.

[19] T. Lau, P. Domingos, and D. S. Weld, "Learning programs from traces using version space algebra," in *Proceedings of the 2Nd International Conference on Knowledge Capture*, ser. K-CAP '03. New York, NY, USA: ACM, 2003, pp. 36–43.

[20] R. Balaniuk, "Drill and join: A method for exact inductive program synthesis," in *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Proietti and H. Seki, Eds., vol. 8981. Springer, 2014, pp. 219–237.

[21] C. Lattner and 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, Mar 2004.

[22] C. Cadar, D. Dunbar, and 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*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

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

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

[25] 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*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[26] R. Brummayer and 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*, ser. TACAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 174–177.

[27] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, Mar. 2012.

Appendix

In the Section we present the full details of the algebraic simplification method used in Section **??**.

The first task consists in identifying the bitwise expressions used in the linear MBA identity. We consider then equivalent decompositions of the full expression, so that we can set some elements of this decomposition. Using these specific elements, we then compute the linear MBA identity, and the polynomial of degree d used in the construction of the expression. The rest of this section deals with the validation of the decomposition, the computation of the constant hidden in the MBA obfuscated conditional. We finally validate the result against the original expression.

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 **??** 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 $f_d(k)$.

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

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

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.

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

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:

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

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 $\alpha_i = \beta_{i,1}/\beta_{i_0,1}$.

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.

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

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

For each $j \in \{0,\dots,d\}$, 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:

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

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

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 **??**. An alternative way is to use the generation algorithm given for such linear MBA identities: the linear MBA identities given in [**?**] 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

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