Indistinguishable Predicates: A New Tool for Obfuscation

Lukas Zobernig, Steven D. Galbraith and Giovanni Russello
The University of Auckland
Email: {lukas.zobernig, s.galbraith, g.russello}@auckland.ac.nz

***Abstract*—Opaque predicates are a commonly used technique in program obfuscation, intended to add complexity to control flow and to insert dummy code or watermarks. We survey a number of methods to remove opaque predicates from obfuscated programs, hence defeating the intentions of the obfuscator. Our main contribution is an obfuscation technique that introduces opaque constant predicates that are provably indistinguishable from obfuscations of certain other predicates in the program. Our technique resists all known efficient static attacks on opaque predicates. We present an evaluation of our implementation of the scheme. This includes measurements of its performance impact on an obfuscated instance versus a vanilla one and an experimental verification that the obfuscator is functionality preserving.**

I. Introduction

Program obfuscation is a general tool used to protect *intellectual property* (IP) from reverse engineering. This paper is concerned with obfuscating predicates. Opaque predicates [1], [2], [3], [4], [5] are commonly used to add complexity to control flow and to insert dummy code or watermarks. For example, a reverse-engineer may compute the *control flow graph* (CFG) of a program and then try to deduce something about the structure of the program from this information. Opaque predicates are constant predicates (always true or always false) that have been obfuscated with the intention of hiding the fact they are constant. One can add complexity to the CFG by introducing opaque predicates that appear to create extra branches and program blocks, even though these blocks are never executed when the program is run.

A major problem with current software obfuscation schemes that protect compiled applications is that they are not secure against automated attacks. Typical schemes that complicate the CFG can be broken by using static analysis [5], [6], [7], [8], [9], [10], [11]. Other obfuscation approaches based on virtual machines are readily defeated by employing advanced dynamic analysis and symbolic execution techniques [12], [13]. In case all automated attacks fail, an adversary could always try to manually deobfuscate programs by employing enough reverse engineering *manpower*. Thus it is interesting and important to build and improve on existing schemes to make them more resilient against deobfuscation by both static and dynamic approaches.

A. Our Contribution

We aim to improve the existing obfuscation approach of using constant predicates to introduce complexity into a program's control flow graph. First, we survey known attacks that can determine whether a given obfuscated predicate is constant or not. We present new attacks that detect constant predicates using approaches that have been considered for deobfuscation before, such as taint analysis. In this work, we mainly consider static program analysis. Once a constant predicate is identified, the control flow of the obfuscated program can be simplified, and any blocks of dummy (non-executed) code can be removed. Second, we present an obfuscation technique that creates opaque (constant) predicates that are indistinguishable from obfuscations (”dressed”) of real predicates in the program. This means it is infeasible for an adversary to distinguish between originally existing predicates in a program and the ones we introduce in the obfuscation step.

Our solution is applicable when obfuscating programs or program segments that contain a large number of constant comparisons (“if $x = c$” where x is a variable and c a constant) or variable comparisons (“if $y = ax + b$” where x and y are variables). These are very common code constructs and we have found libraries such that around 80% of predicates are of this form. Our tool obfuscates these predicates and also inserts opaque predicates (obfuscated constant predicates) to complicate the control flow. The main security property of our tool is that the real predicates and opaque predicates are indistinguisghable under automated static attacks. We give both theoretical and experimental evidence to support our security claim.

We have implemented our proposed obfuscation scheme on top of the LLVM compiler infrastructure as a generic compiler plugin. This allows our obfuscator to target the many different high level programming languages that compile to LLVM bitcode, such as C and C++ for example. It further allows us to support all the different processor architectures that LLVM compiles to, such as x86(_64) and ARM(64) to name a few. We use our LLVM based implementation of the proposed scheme to show how it affects the execution performance of selected examples of open source code.

B. Outline

The remainder of this work is structured as follows. Section II presents related work and other program obfuscation approaches. In Section III, we present our attack model and a list of static detection methods for constant predicates. Section IV formalises the different classes of predicates we will use throughout this work. In Section V, we construct a special class of constant predicates and explain our construction of transforming existing predicates in a program to appear to be of the same form as our constant predicates. Finally, in Section VI, we close the gap between constant predicates and obfuscation. Section VII presents a discussion of how to improve our approach in case of a dynamic attacker setting. In Section VIII, we present our current implementation and in Section IX, we discuss the performance impact of our obfuscation scheme when applied to real code. Section X concludes our paper and presents future research directions.

II. Related Work

In this section we survey the literature on obfuscation. On the one hand, transformations to various levels of a program's representation are applied [14], [15], [16]. Obfuscation is possible on a source code or an intermediate language level. Alternatively, one can obfuscate the processor instructions that a compiled program consists of. The goal of transforming existing program instructions is to hide the underlying functionality. This means that an *addition* might for example be rewritten as a *subtraction* of a negative number. An *exclusive or* operation could be written as a series of *not and* statements. It is possible to extend this basic idea to be applied to more complex statements, such as transforming a mathematical formula to a different and ideally more complex one that exhibits the same functionality [17]. This idea captures the general idea of most obfuscation schemes in use today. Additionally, we can also consider instructions that modify themselves at runtime to change their functionality [18].

Instead of transforming existing program instructions, another strategy is to introduce superfluous or inert instructions interleaved with the original ones. Special care has to be taken such that the new instructions do not interfere with the original computation. This opens the schemes up for different types of attacks that filter the superfluous instructions from an obfuscated sequence by means of dynamic program analysis and taint tracking [6], [19], [20], [21]. These attacks seek to remove the added instructions and restore the original sequence. Other dynamic approaches are based on symbolic or *concolic* (concrete+symbolic) execution using SMT solvers [22], [23]. Certain obfuscation schemes are weak against symbolic execution. To counteract this, control flow loops are transformed in a way that increases symbolic execution time.

Yet another method is concerned with execution flow through a program's control flow graph [24]. The considered approaches range from introducing superfluous control flow [25], [26] to flattening control flow which rewrites a control flow graph into a state machine [27]. In general, a control flow transformation always needs to be chosen in a way that the original control flow is contained in the newly generated control flow graph while introducing artificial complexity. The problem with these techniques is that they are susceptible to automated deobfuscation techniques [6]. Control flow flattening for example can be combatted by using dynamic analysis to identify the state variable and after that rebuilding the original control flow graph by interpreting the state machine.

As we have already stated, another possibility to introduce additional superfluous control flow into a program is by using obfuscated constant predicates. A further application of constant predicates is to encode watermarking information in the program code. This can be utilised by a vendor to distribute unique instances of a compiled program to different consumers. In the event of an unauthorized redistribution of commercial software, watermarking can thus be helpful to uniquely identify the initial source.

Another heavily used technique is *virtualisation* which rewrites existing processor instructions into instructions that are executed by a virtual processor [7], [12], [28], [29], [30]. This processor is then either emulated by interpreting the virtual instructions or by just-in-time compilation and execution on the target architecture. The latter solution is very similar to what existing platforms like *Java* and *.NET* are doing for optimising execution speed. However, obfuscators based on virtualisation have been completely broken by dynamic analysis and taint tracking to extract the embedded functionality in terms of target processor instructions [13], [31].

III. Attacks on Opaque Predicates

We now survey techniques to detect obfuscated constant predicates. The possible methods involve human interaction as well as automated algorithmic interaction [2], [5].

Let $P : X \to \{0,1\}$ be a predicate (here the element 1 will usually represent the Boolean *true* and the element 0 the Boolean *false*), computed in an obfuscated program segment, where X is the predicate's domain. We wish to have automated tools to determine whether $P(x)$ is constant or not, i.e. whether

$$\forall x \in X : P(x) = p$$

for p constant. If we can solve this problem, then we can build an automated reverse engineer tool that takes an obfuscated program, enumerates all its predicates, determines which are constant, and then removes the predicates and any non-executed program blocks. By iterating the process the adversary can try to recover the original version of the program or a close version of it.

We shall first consider a static attacker that does not execute the program. Later in the section we consider dynamic attacks.

A. Brute Force Search

If X is a small enough set that one can efficiently execute the program $P(x)$ for all $|X|$ possible values $x \in X$ then one can easily check if $P(x)$ is an obfuscated constant predicate. For example, if x is a 32-bit word then this requires 2^{32} executions of the program segment, which is non-trivial but feasible. On the other hand, if x is a 64-bit word then this requires 2^{64} executions of the program segment and this is probably more work than one wants to spend on a simple reverse-engineering task. Here we are assuming that the running time of $P(x)$ is more-or-less constant. However, the task may be easier if the value $P(x)$ can already be computed more quickly on some large subset of inputs.

TABLE III.1: List of constant predicates often found in literature and obfuscation solutions. These predicates have been constructed to always evaluate to the same result independent of the input value. Here the value x is usually considered as an unsigned integer of a fixed bitlength or as an element of $\mathbb{Z}/2^n\mathbb{Z}$.

$7y^2 - 1 \neq x^2$
$2\|x(x+1)$
$3\|x(x+1)(x+2)$
$x^2 > 0$
$7x^2 + 1 \not\equiv 0 \mod 7$
$x^2 + x + 7 \not\equiv 0 \mod 81$
$x > 0$ for $x \in I$ random where $I \subset X \setminus \{0\}$ a random interval

As an example susceptible to a brute force attack, we consider the predicate $P(x) =$ "$2|x(x+1)$" where x is an 8-bit byte interpreted as an unsigned integer in $[0, 255]$. This predicate tests whether $x(x+1)$ is divisible by 2, which is always true. A brute-force automated tool can obviously try all x and determine that the predicate is constant.

B. Probabilistic Check

Instead of trying all $x \in X$, one could choose a number of randomly chosen $x \in X$ and execute the program segment to compute $P(x)$ for all these values. If the output is always the same then one might suspect that P is a constant predicate and hence flag it for removal from the program.

The danger is that the program segment may be an obfuscation of a valid predicate that is "mostly" constant (examples of such predicates frequently occur in programs, such as error handling, loop termination conditions, and evasive functions like password checks). Hence this approach is risky. But it is sensible as a pre-processing before applying more sophisticated methods to determine if a predicate is constant or not.

C. Pattern Matching

We have surveyed the literature [1], [3], [32], [33], as well as studied samples of code produced by both free and commercial obfuscation solutions, to collect specific proposals for obfuscated constant predicates. Surprisingly, there are relatively few predicates that are used over and over again. TABLE III.1 list the most-used constant predicates.

One immediately realises that this leads to a possible dictionary attack, where one takes obfuscated predicates from the program being attacked and pattern-matches the source code against example code for the predicates in TABLE III.1. This attack has been mentioned in [1], [3] in the context of removing watermarking.

As an example, we consider the open source implementation of *Obfuscator-LLVM* [33]. It uses a unique static constant predicate $P(x, y) = y < 10 \vee 2|x(x+1)$ where x and y are global program variables. In this case we can simply apply pattern matching to the instructions to detect all opaque predicates in the obfuscated program. Once we have detected such a constant predicate, we are able to clean up the control flow graph by removing the predicate and the superfluous execution path. Hence we see that this pattern-matching attack already defeats (very efficiently) the use of opaque predicates in most obfuscators in the open literature.

Of course, it is easy to create additional constant predicates that are not listed in TABLE III.1, but this does not seem to have been done in any large-scale way in current obfuscation solutions. One can also use the approach in [1] to introduce a class of constant predicates that is parametrised by some parameter n (a multiple of an algebraic identity for example). Even though this methods yields a large set of different constant predicates, it is still possible to detect them using a pattern matching approach, so the attack is still powerful.

Pattern matching is partially independent of the processor architecture. A first approach is to implement the matching for certain common processor architectures that known obfuscation solutions target. In a generalisation of this approach, the matching rules are given in terms of a higher level intermediate language such as LLVM for example. In this more general case, translators from different processor architectures to the intermediate language are required. The advantages are that future matching rules can be given in terms of the intermediate language and new processor architectures are readily supported by implementing new translators. This way a new architecture automatically supports all the existing known constant predicates already.

D. Automated Proving

Another approach to determining if a program segment computes a constant predicate is to run an SMT-solver. We shall call an obfuscated predicate $P : X \to \{0, 1\}$ SMT-solvable if a SMT solver is able to efficiently answer whether P is constant or not. It is clear that this strongly depends on the size of the space X and the *complexity* of P.

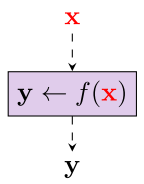
As an example consider again the predicate $P(x) =$ "$2|x(x+1)$" where X is some subset of $\mathbb{Z}$. A human or automated solver could come up with a proof that the predicate is constant by expanding $P(x)$ for $x = 2k$ and $x = 2k+1$, where $k \in \mathbb{Z}$. We verify that $P(2k)$ is always true and $P(2k+1)$ is always true.

E. Taint Analysis

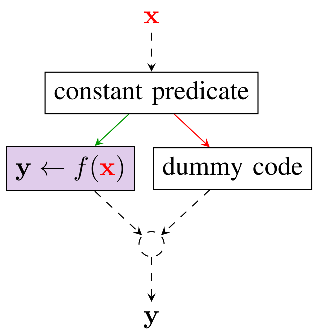
Instead of considering the code for a predicate in isolation we can look at the code blocks that are selected by the conditional expression. Fig. 1 presents in (a) a code block and in (b) an obfuscated constant predicate (that is always true) and a code block ("dummy code") that is never executed. If the "dummy code" is chosen poorly then there may be no data dependency on the input variables of an obfuscated function. Taint analysis [21] can then be used to identify code blocks with no data dependencies. Hence, one can use taint analysis to flag predicates as being potentially constant. This is a more dynamic type of attack than considered in the earlier sections.

F. Execution Traces

This is essentially a dynamic version of the probabilistic check mentioned in Section III-B. One executes the obfuscated



(a) Original input control flow graph. The basic block produces an output $\mathbf{y}$ that depends on the input $\mathbf{x}$.



(b) Control flow graph obfuscated by introducing a constant predicate. Note that the inserted basic block does not depend on the input $\mathbf{x}$.

Fig. 1: Example of extracting the original CFG from obfuscated CFG using taint analysis. The nodes that have no data dependence on the input can be ignored when extracting the logic that operates on the input.

program in a debugger or other controlled environment and records the computed values of all predicates. Since the predicates are being evaluated on actual executions of the program, it is possible to identify nearly-constant predicates such as loop terminations. As with the probabilistic check, this approach allows to efficiently flag certain predicates as being opaque, but one cannot be certain that the program obtained by removing all such predicates is error free.

IV. Classes of Predicates

For future reference we briefly introduce some terminology that is relevant in our discussion of predicates.

Definition 1. *A predicate P on a set X is called evasive if*

$$\mathbf{Pr}_{x \leftarrow X}[P(x) = 1] \leq \tfrac{c}{|X|}$$

for some small constant c.

In other words, a predicate is evasive if it is false for almost all inputs $x \in X$.

In practice one should consider different classes of predicates. A class $\mathcal{C}$ of predicates is evasive if each predicate individually is evasive and if, for each $x \in X$, the probability over all $P \leftarrow \mathcal{C}$ that $P(x) = 1$ is small. An example of an evasive predicate class is the set of password check functions $P(x) =$ "$x == pw''$ over all possible passwords.

Since it is hard to find an input x that satisfies an evasive predicate class, this class of predicates is a good candidate for obfuscation, and there is a large literature on the problem [34], [35].

Definition 2. *A predicate P is called balanced if*

$$\mathbf{Pr}_{x \leftarrow X}[P(x) = 0] = \frac{1}{2}.$$

This means that for a balanced predicate the probability for it to evaluate to either value in $\{0, 1\}$ is the same.

Definition 3. *A predicate P is called noticeable if*

$$\forall p \in \{0, 1\} : \mathbf{Pr}_{x \leftarrow X}[P(x) = p] \geq \frac{1}{\mathrm{poly}(n)}$$

where $|X| = 2^n$.

Noticeable predicates can be efficiently distinguished from constant predicates by the probabilistic check method of Section III-B.

V. Infeasible Predicates

This section describes our main contribution, which is an obfuscation tool for constant and certain evasive predicates. Our obfuscation tool is based on standard cryptographic notions such as hash functions and encryption. We will consider classes $\mathcal{C}$ of predicates and show that no adversary can efficiently determine if an obfuscated predicate in the class is constant or not.

A. Obfuscating Constant Comparison Functions using Hash Functions

It is folklore that one can obfuscate a password check (constant comparison) "$x == pw$" using a cryptographic hash function H by computing $h = H(pw)$ and publishing the obfuscated predicate "$H(x) == h$". This has been considered before to hide code checking for malware triggers [36] for example. We will use this idea to give an obfuscation process, such that no efficient adversary can distinguish whether the obfuscated predicate is a constant predicate or a constant comparison.

The following lemma is a basic tool in our security analysis. We use the notation ϵ for the empty string, and if u, v are binary strings we write $u \| v$ for their concatentation.

Lemma 1. *Let $H : \{0, 1\}^n \to \{0, 1\}^n$ be a cryptographic hash function. For each $1 \leq k \leq n$ let $X_k = \{0, 1\}^k$. Define an oracle O that takes as input $(X_k, t \in \{0, 1\}^{n-k}, y \in \{0, 1\}^n)$ and returns 1 if there exists $x \in X_k$ such that $H(x\|t) = y$ and 0 otherwise. Then given $y \in \{0, 1\}^n$ one can, using polynomially many calls to O, compute some $x \in \{0, 1\}^n$ such that $H(x) = y$ or determine that no such x exists.*

Proof. Calling $O(X_n, \epsilon, y)$, where ϵ is the empty string, decides if x exists or not. If x exists, set $t_0 = \epsilon$ and iterate the following process for $i = 0, 1, 2, \ldots$: Given that $O(X_{n-i}, t_i, y) = 1$ we call $O(X_{n-(i+1)}, 0\|t_i, y)$. If the result is 1 then set $t_{i+1} = 0\|t_i$, else set $t_{i+1} = 1\|t_i$. On termination we set $x = t_n$. □

With this result in hand, here is our construction of an obfuscator. Let $H : \{0, 1\}^n \to \{0, 1\}^n$ be a cryptographic hash function. Let $X = \{0,1\}^k \subseteq \{0,1\}^n$. Let $\mathcal{C}$ be the class consisting of all constant comparison predicates $P(x) =$ "$x == c$" for $x, c \in X$ together with the constant predicate $P(x) = 0$. To obfuscate a comparison predicate "$x == c$" for some $c \in X$ the obfuscator randomly chooses $t \in \{0,1\}^{n-k}$ and computes $C = H(c\|t)$. The obfuscated predicate $P(x)$ computes $y = H(x\|t)$ and then checks if $y = C$.

To make an opaque predicate (obfuscate the constant function) we choose a random $C \in \{0,1\}^n$ and $t \in \{0,1\}^{n-k}$ and publish the obfuscated predicate $P(x)$ that computes $y = H(x\|t)$ and checks if $y = C$. With probability $1/2^{n-k}$ there is no solution $x \in \{0,1\}^k$ to this equation, and so with high probability the predicate is always false.

Theorem 1. *Let $H : \{0,1\}^n \to \{0,1\}^n$ be a hash function such that it is hard to compute pre-images.. Then, there does not exist any efficient adversary that, for all $X = \{0,1\}^k$ and for any obfuscated predicate as above, can determine whether the predicate is constant or not.*

Proof. Let A be an efficient adversary that, for all k, takes an obfuscated predicate on $X = \{0,1\}^k$ and determines if the predicate is constant or a constant comparison. Then A performs the function of the oracle O in Lemma 1. Hence one can use A (executed at most n times) to compute a preimage of H. But this contradicts the assumption that the hash function is preimage-resistent. $\square$

It is natural to use a cryptographic hash function in this construction, but note that we do not require collision resistance for the security result. Hence one can consider less strong hash functions than many other crypto applications (and also shorter output lengths). One can also consider replacing H with a deterministic encryption function.

B. Obfuscation using Homomorphic Encryption

We now consider a different class of predicates, that are still evasive. Let $X = \mathbb{Z}/q\mathbb{Z}$ (for example with $q = 2^n$) and let $k \in \mathbb{N}$. Consider the class $\mathcal{C}_k$ of predicates $P(x_1, \ldots, x_k)$, each corresponding to a vector $(a_1, \ldots, a_k) \in X^k$ that return 1 if and only if

$$a_0 + a_1x_1 + \cdots + a_kx_k \equiv 0 \pmod{q}. \qquad \text{(V.1)}$$

This is a large class of predicates (there are q^{k+1} elements in $\mathcal{C}_k$). The class is evasive: If $(a_0, a_1, \ldots, a_k) \in X^k$ is not known then the probability that a random input $(x_1, \ldots, x_k) \in X^k$ satisfies the predicate is $1/q$. Such predicates have been considered in [35].

An important member of this class is the variable comparison predicate $P(x, y) =$ "$x == y''$. Predicates of this form appear frequently in real programs, and it is valuable to produce tools to obfuscate them. However, it is important to remember that attacks such as those given in Section III can always be used to test whether an obfuscated predicate is of a particular form (for example, executing it on inputs that are known to satisfy the predicate; in this case on pairs (x, x)). Hence the tools in this section are most appropriate when the program naturally contains a rich variety of predicates from the class $\mathcal{C}_k$.

The solution from [35] relies on the discrete logarithm problem. Let g be an element of a group, such that the order of g is q and such that the discrete logarithm problem is hard (hence we require q to be very large). One can publish $b_0 = g^{a_0}, b_1 = g^{a_1}, \ldots, b_k = g^{a_k}$ as the obfuscated predicate. On input $(x_1, \ldots, x_k) \in X^k$ the program computes

$$b_0 b_1^{x_1} \cdot b_2^{x_2} \cdots b_k^{x_k} = g^{a_0 + \sum_{i=1}^k a_i x_i}.$$

If the predicate is true then this is the element $1 = g^0$ in the group, otherwise it is a random group element.

We can extend this approach using homomorphic encryption. Denote by $E_{pk}(m)$ the encryption of a plaintext message m with respect to a public key pk. Here $E_{pk} : X \to Y$ where $(X, +_X, \times_X)$ is the space of all possible messages m and $(Y, +_Y, \times_Y)$ is the space of all corresponding ciphertexts c. Note that we do not necessarily have $|X| = |Y|$, think of a *semantically secure* cryptosystem for example where $|X| \neq |Y|$. E_{pk} satisfies the homomorphism properties

$$\begin{aligned} E_{pk}(m_1) +_Y E_{pk}(m_2) &= E_{pk}(m_1 +_X m_2), \\ E_{pk}(m_1) \times_Y E_{pk}(m_2) &= E_{pk}(m_1 \times_X m_2). \end{aligned}$$

To obfuscate a predicate from the class $\mathcal{C}_k$ one publishes $b_i = E_{pk}(a_i)$ for $1 \le i \le k$. Then to compute the predicate on input $(x_1, \ldots, x_k)$ one computes

$$\sum_{i=1}^k b_i \times E_{pk}(x_i)$$

which will be an encryption of 0 iff the predicate is true. Assuming one can detect encryptions of zero without knowing the private key then this scheme can be used as an obfuscator.

C. Variable Point Comparisons

We return to the class of predicates in equation (V.1). In this section we call them variable point comparisons, since they include special cases such as $P(x, y) =$ "$x == y''$ and $P_r(x, y) =$ "$x \equiv y + r \pmod{q}$". As already mentioned, Canetti et al [35] showed how to obfuscate this class using the discrete logarithm problem. We will now explain how to make constant predicates that are indistinguishable from obfuscations of predicates of this form.

Let q be a large prime and let $X \subseteq \mathbb{Z}$ be a set of size $|X| < q$. Suppose we have a predicate $P(x_1, \ldots, x_k)$ on X^k such that the set of k-tupes $(x_1, \ldots, x_k) \in X^k$ that satisfy the predicate are exactly the k-tuples that satisfy equation (V.1). For example, we might have $X = [0, 2^{64} - 1]$ corresponding to 64-bit words and yet $q > 2^{256}$. To obfuscate the predicate $P(x_1, \ldots, x_k)$ we publish the group elements $b_i = g^{a_i}$ for $0 \le i \le k$ where g is an element of order q in some group such that the discrete logarithm problem is hard. To execute the obfuscated program, as before we compute

$$b_0 \prod_{i=1}^k b_i^{x_i}$$

and check if the value is equal to 1.

Now suppose we want to disguise a constant predicate as being a predicate of this form. We then simply choose random group elements $b_0, \ldots, b_k$ of order q. If $|X|^k < q$ then it is likely that there is no k-tuple $(x_1, \ldots, x_k) \in X^k$ that satisfies Equation (V.1). This means the obfuscated predicate is a constant predicate, but it cannot be distinguished from an obfuscation of a real variable point comparison.

D. Obfuscating Variable Comparison Functions using Hash functions

For a variable point comparison of the form $P(x, y) =$ "$ax+b == y$" we can also employ a hash function to disguise it as a constant predicate. Here a and b are constants. For this let $H : \{0,1\}^n \to \{0,1\}^n$ be a cryptographic hash function. Let $X = \{0,1\}^k \subseteq \{0,1\}^n$. Let $\mathcal{C}$ be the class consisting of all variable comparison predicates $P(x, y) =$ "$ax + b == y$" for $x, y \in X$, $a, b \in X$ constant together with the constant predicate $P(x) = 0$. The obfuscator chooses a random $t \in \{0,1\}^{n-k}$ and a random $r \in X$ and computes the hash $C = H(r\|t)$. The obfuscated predicate $P(x, y)$ computes $h = H(ax + b - y + r\|t)$ and then checks if $h = C$.

For an opaque predicate we choose a random $t \in \{0,1\}^{n-k}$, a random $r \in X$ and a random $C \in \{0,1\}^n$ and publish the obfuscated predicate $P(x, y)$ that computes $h = H(ax + b - y + r\|t)$ and checks whether $h = C$. Then again with high probability this opaque predicate is always false.

VI. Constant Predicates and Obfuscation

Now that we have introduced all required notions concerning constant predicates, we are able to close the gap between constant predicates and obfuscation. For this we first state a formalism of program obfuscation and subsequently state a short definition of the control flow graph and its elements - basic blocks and program instructions. After this introduction we describe how the obfuscation scheme inserts constant predicates into the program and how it dresses existing predicates.

A priori a program obfuscator $\mathcal{O}$ is a mapping that takes a given input input program $\mathcal{P}_{\text{in}}$ and transforms it into an output program $\mathcal{P}_{\text{out}}$

$$\mathcal{O} : \mathcal{P}_{\text{in}} \mapsto \mathcal{P}_{\text{out}}.$$

Here the programs are defined over $\mathcal{P}_{\text{in}}, \mathcal{P}_{\text{out}} : \{0,1\}^n \to \{0,1\}^m$ taking an n-bit input vector and producing an m-bit output vector. Note that we have not made any assumptions about the functionality of $\mathcal{P}_{\text{out}}$. We will construct our obfuscation method such that it preserves the input program's functionality.

Definition 4. *We say that an obfuscator $\mathcal{O}$ is functionality preserving if $\mathcal{O}$ satisfies*

$$\forall \mathbf{x} \in \{0,1\}^n : \mathbf{Pr}[\mathcal{O}(\mathcal{P}_{\text{in}})(\mathbf{x}) = \mathcal{P}_{\text{in}}(\mathbf{x})] = 1 - \epsilon$$

where ϵ is negligible.

This means that the program returned by a functionality preserving obfuscator has a negligible possibility of producing a different result than the non-obfuscated program.

A. Control Flow Graph

We assume that a general program is made up of many smaller building blocks, namely functions. In the following we will focus on obfuscating the *control flow graph* (CFG) of a function. We closely follow LLVM's definition of the CFG [37]. The CFG $G = (V, E)$ is the graph consisting of the set of all *basic blocks* V and the set of all *control flow edges* $E \in V \times V$ of a program. A basic block $B_i \in V, i \in I$ is an ordered tuple $B_i = [\beta_j]_{j \in J}$ of *program instructions* β_j. A control flow edge can also be represented by the ordered pair (i, j) with $i, j \in I$ modelling the control flow transfer from basic block B_i to B_j.

Note that a basic block can have multiple predecessors and multiple successors. The control flow in a basic block is linear. A control flow transfer can only possibly happen with the last program instruction in a basic block. Thus basic blocks can be considered as the basic building blocks of a function.

A program instruction β_j generally models the assignment of register or memory locations with the result of a function applied to several values taken from registers or memory locations. We shall denote this by writing

$$\mathbf{y} \leftarrow \mathbf{F}(\mathbf{x})$$

where $\mathbf{x}$ is the vector of inputs and $\mathbf{y}$ is the memory of assigned outputs. Additionally, there exists a special class of instructions, namely those that result in control flow transfers. These *branch* instructions terminate the basic block tuple of instructions and can never appear in any other position. A branch may additionally depend on the output value of a predicate $y = P(\mathbf{x})$. In the unconditional case we denote the branch by $\mathbf{B}(B_t)$ with B_t the branch target. In the conditional case we write $\mathbf{BCOND}_y(B_0, B_1)$ which results in a branch to the target block B_y with $y \in \{0,1\}$.

B. Obfuscation

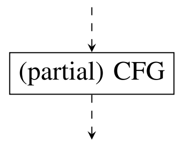
Using constant predicates, the idea is now to introduce superfluous control flow into an existing control flow graph. In the simplest possible model we take a full or partial CFG and prepend it with a constant predicate. An example for a basic block modelling this is given by

$$B = \begin{bmatrix} y \leftarrow \mathbf{CONSTP}(\mathbf{x}) \\ \mathbf{BCOND}_y(B_t, B_o) \end{bmatrix}.$$

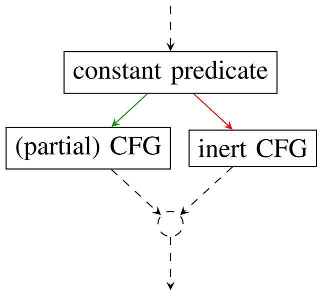
Here $\mathbf{CONSTP}(\mathbf{x})$ computes a constant predicate $P(\mathbf{x})$. The entry block of the original CFG is given be B_t and the entry block of the inserted inert CFG is given by B_o.

The inserted inert CFG can be generated of arbitrary complexity as it will never be executed. The situation is depicted in Fig. 2. Instead of merging the paths, it is also possible to have the inert path finally branch to any other basic block in the CFG or back to the basic block evaluating the constant predicate to form a loop.

A more advanced approach is to dress existing predicates as constant predicates to harden the obfuscated program against pattern matching. In Section V we have described schemes that allow us to dress point comparison predicates



(a) Input control flow graph that is to be obfuscated.



(b) Output control flow graph after inserting a constant predicate and random superfluous code.

Fig. 2: Obfuscating control flow graph using a constant predicate. The input is prepended by a constant predicate and random superfluous code is inserted in the branch that is never taken.

as constant predicates. By doing so a static attacker will not be able to distinguish between the original predicate and an injected constant one. Consider a constant point comparison $P(x)$ = "$x == c$". Our obfuscation scheme $\mathcal{O}$ transforms this predicate according to

$$\begin{bmatrix} x \leftarrow \cdots \\ y \leftarrow \mathbf{CMP}(x, c) \\ \mathbf{BCOND}_y(B_0, B_1) \end{bmatrix} \overset{\mathcal{O}}{\mapsto} \begin{bmatrix} x \leftarrow \cdots \\ h \leftarrow \mathbf{H}(x) \\ y \leftarrow \mathbf{CMP}(h, h_c) \\ \mathbf{BCOND}_y(B_0, B_1) \end{bmatrix}$$

where $h_c = H(c)$ the hash of the constant c and $\mathbf{CMP}(a, b)$ is the operation that compares a and b and returns true or false accordingly.

Analogously we define the process for variable point comparisons following the construction of Section V-D. Suppose a variable point comparison $P(x, y)$ = "$x == y$". To dress it, we generate a random integer r and its hash $h_r = H(r)$. The comparison predicate is the dressed according to

$$\begin{bmatrix} x \leftarrow \cdots \\ y \leftarrow \cdots \\ z \leftarrow \mathbf{CMP}(x, y) \\ \mathbf{BCOND}_z(B_0, B_1) \end{bmatrix} \overset{\mathcal{O}}{\mapsto} \begin{bmatrix} x \leftarrow \cdots \\ y \leftarrow \cdots \\ h \leftarrow \mathbf{H}(x - y + r) \\ z \leftarrow \mathbf{CMP}(h, h_r) \\ \mathbf{BCOND}_z(B_0, B_1) \end{bmatrix}.$$

Any other constant predicate that we might introduce into the CFG should then ideally be of the same form. Note that this way of introducing constant predicates avoids the pattern-matching attack from Section III-C: an attacker cannot remove all predicates that "look like" constant predicates, as some of them are real comparisons and their removal will not maintain correctness of the program. This implementation does not use the randomiser t which was defined in the schemes proposed in Section V as our evaluation implementation will make use of 64-bit values and a 64-bit hash function.

We also have to protect our scheme against taint analysis, an approach we described in Section III-E. To mitigate the attack, we have to introduce a dependency on the input variable $\mathbf{x}$ in the inert CFG in Fig. 1b. This way both possible execution paths will depend on $\mathbf{x}$ and an adversary will not be able to discard the superfluous path without having to solve the predicate.

In Fig. 3 we can see how such a superfluous statement has been introduced in the path that is never taken. This statement depends on the input variable $\mathbf{x}$. A priori the statement could produce an output variable vector $\mathbf{y}' \neq \mathbf{y}$ that is not equal to the output of the original path. Yet to protect the scheme against taint analysis that traverses the CFG in the reverse direction, the statement needs to produce $\mathbf{y}$ or at least feature $\mathbf{y}' \subset \mathbf{y}$.

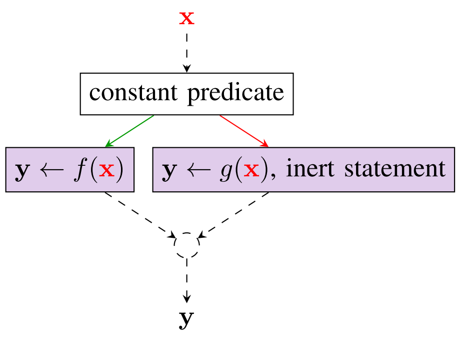

$\mathbf{x}$
constant predicate
$\mathbf{y} \leftarrow f(\mathbf{x})$
$\mathbf{y} \leftarrow g(\mathbf{x})$, inert statement
$\mathbf{y}$

Fig. 3: Obfuscated control flow graph with both branches showing a data dependency on the input variable $\mathbf{x}$. This is done to mitigate against data flow analysis being able to detect the superfluous branch.

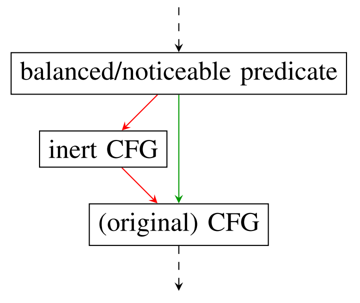
VII. Discussion

In this section, we present further ideas for improving our obfuscator against a dynamic adversary and a possible application of the obfuscator for program watermarking.

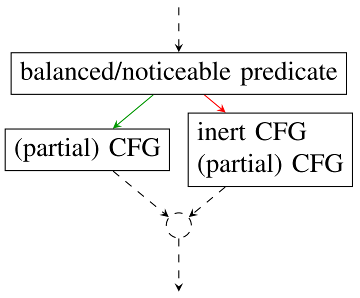
A. Protection Against Dynamic Analysis

In our approach we use infeasible constant predicates and dressed predicates to obfuscate a program that can resist deobfuscation against static analysis methods. Given the way in which we dress original predicates, an adversary is not able to distinguish between original and inserted predicates. Using an SMT solver will not help an adversary to identify the constant predicates our obfuscator inserts. Moreover, because the constant predicates have data dependences on input variables used in the original program, they also evade detection using taint analysis.

Here, we briefly address the case in which an adversary employs dynamic program analysis such as symbolic execution and taking program traces for random inputs. While in this setting infeasibility is still a strong protection against



(a) The balanced/noticeable path branches to the original CFG.



(b) The original CFG was appended to or interleaved with the inert CFG.

Fig. 4: A control flow graph is obfuscated using a balanced/noticeable predicate.

automated solving for constancy, an attacker might alternatively employ other means of extracting the original code paths. The adversary might run the obfuscated program under a probabilistic detection method to detect all the predicates that show a constant behaviour up to some probability. Removing these (almost) constant predicates would give the adversary a simplified CFG close to the original program CFG. The assumption here is that even if some of the removed predicates are dressed predicates, these might be used in the original program for exceptional cases (e.g., error and exception handling branches). Therefore, the adversary would still be able to deobfuscate the main functionality of the program.

A possible solution to this is to introduce balanced or noticeable predicates instead of constant ones. Fig. 4 shows two cases on how to use balanced/noticeable predicates to obfuscate the CFG of a program. The new path is constructed by appending the original CFG to an inert CFG or interleaving both of them. We could also modify the original path to contain superfluous instructions. Although not shown in the figure, the original predicates are also dressed to look like the inserted ones.

Using this technique, we are able to obfuscate the CFG of the original program to resist a dynamic adversary as long as the behaviour of the inserted predicates is indistinguishable from the original predicates. To achieve this, the balanced/noticeable predicates need to exhibit the same behaviour as the dressed predicates. For an explicit construction we can for example use a correctly tuned evasive function as described in Section V-A.

Note that it is important to construct the inert path in a way that it does not affect the original execution behaviour. Yet, to mitigate the possible use of taint analysis as described in Section III-E, we have to make sure it also depends on the function's input variables. The question of how to construct an inert yet intelligible CFG needs to be looked at carefully in the future. The instructions need to be generated in a random fashion such that pattern matching is not possible on any abstraction level. Furthermore, the instructions need to exhibit a data dependency on the input such that taint analysis cannot simply identify them as *dead code*. Ideally, the output data should also depend on the inert instructions in a non-trivial yet identity preserving way. This means that we need transformations of the form $\mathbf{y} \leftarrow F(\mathbf{x})$ that are non-trivial yet may be replaced by a simple $\mathbf{y} \leftarrow \mathbf{x}$. From an intuitive point of view such a function could be created by dressing the identity function as an evasive function.

B. *Program Watermarking*

An interesting question is reproducibility of the compilation step. While our implementation uses a generic random number generator, it is possible to switch to a seeded cryptographic number generator. For the same random seed, the obfuscated binaries would then be identical provided the same compiler version is used. One could also imagine to encode special information in the distribution and parameters of the dressed and constant predicates. This would make the scheme applicable to software watermarking as described in [1].

VIII. Implementation

We have implemented our proposed obfuscation scheme as a plug-in on top of the LLVM compiler infrastructure [37], [38]. LLVM offers an excellent set of tools that support several programming languages and can compile for different hardware architectures. Our obfuscator is implemented in C++ and consists of approximately 500 lines of code (LoC). Integrating the obfuscator into the LLVM toolchain makes our tool language- and architecture-agnostic.

Fig. 6 provides an overview of how our plug-in integrates into the LLVM pipeline. To obfuscate a target program with our obfuscator, first the input source code is compiled into LLVM intermediate language. The obfuscator then processes the intermediate representation (IR) of the target program operating as an LLVM optimiser. Finally, the obfuscated IR is then compiled for the specified architecture.

To see how our obfuscator transforms the control flow graph of a function, we consider the simple function listed in Fig. 5. Fig. 9 shows the graphical representation of the CFG corresponding to Fig. 5 before (Fig. 9-(a)) and after the obfuscation pass (Fig. 9-(b)). From the figure, one can appreciate *visually* how the obfuscated CFG appears to be more *complex* and that its structure is randomised in comparison to the original input.

Fig. 7 lists the decompiler-generated output of the compiled binary created from the source code in Fig. 5. We note that the decompiler was able to reproduce the original source

```
1  int64_t foo()
2  {
3      int64_t sum = 0;
4      for(int64_t i = 0; i != 20000; i++)
5      {
6          if(i % 2)
7              sum += i;
8      }
9      return sum;
10 }
```

Fig. 5: The source code of a sample function that returns the sum of all even integers $i \in [0, 20000)$.

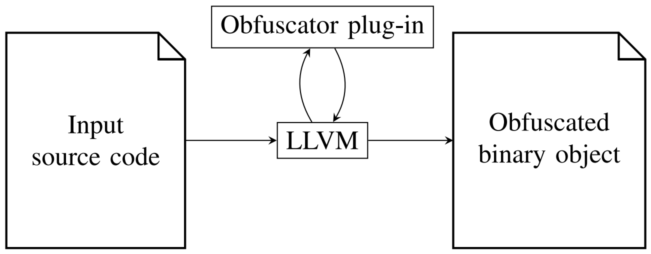


Fig. 6: Overview of the intgration of our obfuscator plug-in in the LLVM pipeline.

code very closely. Fig. 8 lists the decompiler-generated output of the obfuscated binary generated from the source code in Fig. 5. Due to the dressing of the existing predicates and the additional constant predicates, the decompiler was not able to generate any immediately meaningful source code. In particular, lines 9-10 in Fig. 8 are an example of a constant comparison predicate as described in Section V-A. Lines 24-25 in Fig. 8 are one instance of a variable comparison predicate as described in Section V-D. Due to our construction, it is infeasible to tell whether they are dressed or inserted constant predicates.

Note that given the original source, we are able to match the predicate of line 6 in Fig. 5 with lines 20-22 in Fig. 8. However, we cannot immediately match the loop predicate of line 4 in Fig. 5 as easily. This further shows that our obfuscation solution is useful to protect against adversaries that have access to advanced static reverse engineering tools such as decompilers.

Currently, our implementation applies our obfuscation technique to each compiled function, dressing every dressable predicate and inserting a constant predicate in every control flow edge. Although this approach maximises the robustness of the obfuscation, it also represents the worst case in terms of a possible performance penalty. One optimisation that we could easily implement is to let the developer decide which functions should be obfuscated by leveraging LLVM's annotation metadata. Moreover, we could let the developer specify the number of constant predicates that should be inserted and which fraction of dressable predicates should be dressed.

In Section V, we have established that a cryptographic hash function needs to be used in a secure real-world implementa-

```
__int64 foo()
{
  /* ... */
  v3 = 0LL;
  for ( i = 0LL; i != 20000; ++i )
  {
    *(_QWORD *)&v0 = i;
    *((_QWORD *)&v0 + 1) = (unsigned __int128)i
        >> 64;
    if ( (unsigned __int64)(v0 % 2) )
      v3 += i;
  }
  return v3;
}
```

Fig. 7: Decompiler output of the non-obfuscated version of Fig. 5.

tion. In our implementation, we have relaxed this requirement and decided to use the 64-bit version of the *FNV hash* [39]. We have experimentally tested the robustness of this hash function using the SMT solver Z3 [40] to invert the hash of the constant value 0: After 72h of runtime the solver did not manage to succeed. We concluded from this test that the use of the *FNV hash* is quite adequate for the purpose of our prototype.

IX. Performance Evaluation

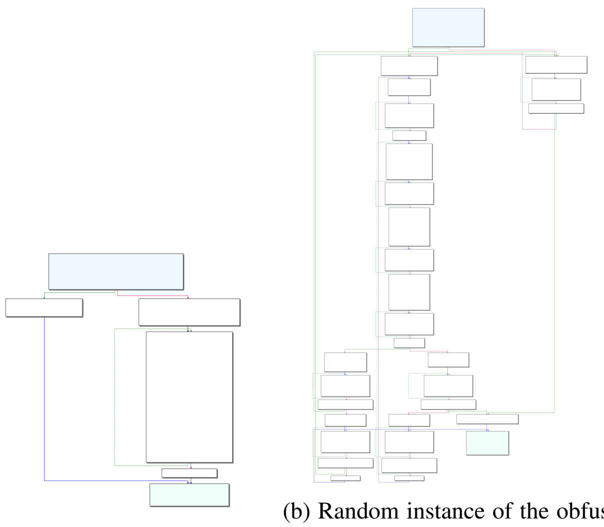
In this section, we analyse the experimental results we collected using the implementation of our obfuscator. The following experiments were conducted on a test machine running Ubuntu 17.04 64-bit. The hardware setup consisted of 32 GB RAM and an Intel(R) Core(TM) i7-4770 CPU clocked at 3.40GHz. The obfuscator was present as a LLVM optimizer plug-in on top of LLVM/Clang version 4.0.0. In all experiments, all dressable predicates have been transformed and a constant predicate was introduced for each existing edge in the input CFG. As a result, the number of edges in the output CFG is doubled when compared to the CFG of the original program.

As benchmark programs for our tests, we have chosen to obfuscate two open source cryptographic libraries: [1]OpenSSL 1.1.0f and [2]mbed TLS 2.5.1. The reasons behind our choice can be explained as follows. First of all, OpenSSL was already used for testing the performance of the Obfuscator-LLVM presented in [33]. This gives us a baseline to compare our approach to. Second, both libraries are large software projects deployed in real-world applications and feature a reasonable variety of program constructs. Third, these libraries ship with self-testing and benchmarking logic: the testing logic allows us to verify that our obfuscator generates code that is functionally equivalent to the original code; the benchmarking logic enables us to measure the performance impact introduced by our obfuscator. The opaque predicates inserted by our obfuscation tool are constant with very high probability, but there is the potential for errors to occur at runtime if some predicate turns

[1]https://www.openssl.org/source/openssl-1.1.0f.tar.gz
[2]https://tls.mbed.org/download/mbedtls-2.5.1-apache.tgz

```
__int64 foo()
{
  /* ... */
  v14 = 0LL;
  v13 = 0LL;
  while ( 1 )
  {
LABEL_2:
    LODWORD(v0) = fnv64_u64(v13);
    if ( v0 == -6175153156727064853LL )
    {
      fnv64_u64(v13);
      return v14;
    }
    LODWORD(v6) = fnv64_u64(v13);
    if ( v6 == 7014728644095366902LL )
      goto LABEL_18;
    *(_QWORD *)&v1 = v13;
    *((_QWORD *)&v1 + 1) = (unsigned
        __int128)v13 >> 64;
    v12 = v1 % 2;
    LODWORD(v2) = fnv64_u64(v12);
    if ( v2 == -6284781860667377211LL )
      break;
    LODWORD(v7) = fnv64_u64(v13 - v12 + 23);
    if ( v7 != 5761928859755592534LL )
      goto LABEL_6;
LABEL_18:
    while ( 1 )
    {
      LODWORD(v10) = fnv64_u64(-329LL);
      if ( v10 == 6784497596726496898LL )
        break;
      v4 = v13++;
      LODWORD(v11) = fnv64_u64(v4 - v13 + 111);
      if ( v11 != 1874020204673976065LL )
        break;
LABEL_6:
      v3 = v14;
      v14 += v13;
      LODWORD(v9) = fnv64_u64(v3 - v13 + 7);
      if ( v9 == 9000058536377130811LL )
        goto LABEL_2;
    }
  }
  LODWORD(v8) = fnv64_u64(v13 - v12 + 118);
  if ( v8 != 3035873277477129725LL )
    goto LABEL_18;
  return v14;
}
```

Fig. 8: Decompiler output of the obfuscated version of Fig. 5.



(a) Input CFG.

(b) Random instance of the obfuscated CFG.

Fig. 9: Example of input and a random output by applying our implementation of the proposed obfuscation method. The entry respectively exit blocks have been shaded.

TABLE IX.1: Vanilla OpenSSL vs. obfuscated OpenSSL self-test results. The vanilla reference version passes all 548 tests. The obfuscated version also passes all 548 tests. Note that the time required for the obfuscated version to finish all tests is roughly 4 times the time it took the vanilla version.

OpenSSL Type	# Tests passed (of 548)	Running time
Vanilla	548	23s
Obfuscated	548	80s

out to be non-constant. Hence we have used the self-testing suites to confirm the correct execution of our obfuscated programs.

A. *OpenSSL 1.1.0f*

OpenSSL is an open source collection of routines implementing the TLS (Transport Layer Security) and SSL (Secure Socket Layer) protocols. It consists of roughly 470K LoC and is widely used in different Unix-like operating systems distributions [41]. The main application of OpenSSL is to secure and encrypt the network communication between web clients and web servers. In addition to that, it provides generic access to various symmetric and asymmetric cryptographic algorithms such as encryption, secure hashing and large integer arithmetic to name a few.

In our analysis, out of 36855 integer comparisons present in the library a total of 28890 (78.4% of the total predicates) resulted as constant or variable point comparisons that our obfuscator dressed as infeasible predicates. Moreover, a total of 88458 infeasible constant predicates were inserted.

After the code was obfuscated, we executed the tests included with the library for both the vanilla and obfuscated versions. From TABLE IX.1, we can see that the obfuscated version of OpenSSL passes all self-tests in roughly 4 times the amount of time that the vanilla version requires.

Next, we executed the benchmark tests for the symmetric cryptographic algorithms for both the vanilla and obfuscated versions. OpenSSL's benchmark operates on block sizes ranging in 16, 64, 256, 1024, 8192 and 16348 bytes. For each of these block sizes, it produces a mean performance value by executing the symmetric algorithms for one block size multiple times. We have computed the mean performance for each algorithms over all the block sizes along with the standard deviation, shown on Fig. 10. In Fig. 11, we can see the performance impact that the obfuscation has on the individual symmetric algorithms. As we can see, the performance of the obfuscated version is not too far from the vanilla version. This

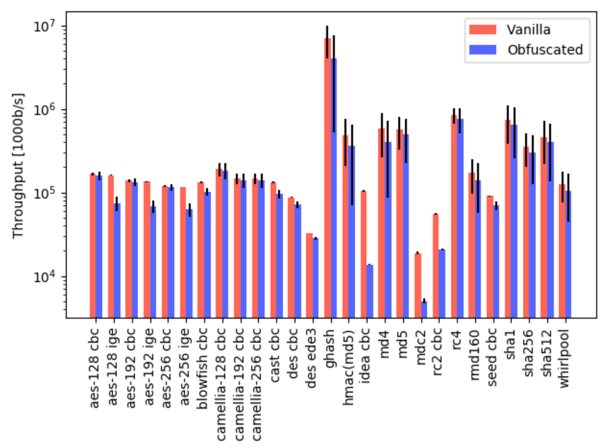


Fig. 10: OpenSSL 1.0.0f vanilla vs obfuscated symmetric algorithms benchmark results. The vertical axis is scaled logarithmically and shows the throughput in 1000 bytes per second. The horizontal axis denotes the different algorithms.

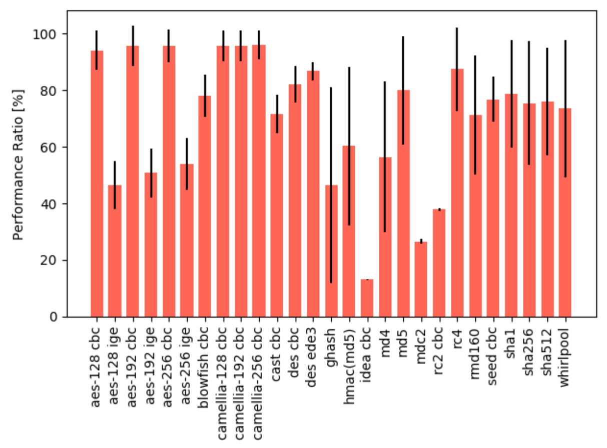


Fig. 11: OpenSSL 1.0.0f vanilla vs obfuscated symmetric algorithms performance. The vertical axis shows the performance ratio of the obfuscated version of each algorithm as a percentage of the vanilla version. The black bars show the standard deviation of the percentage. The horizontal axis denotes the different algorithms.

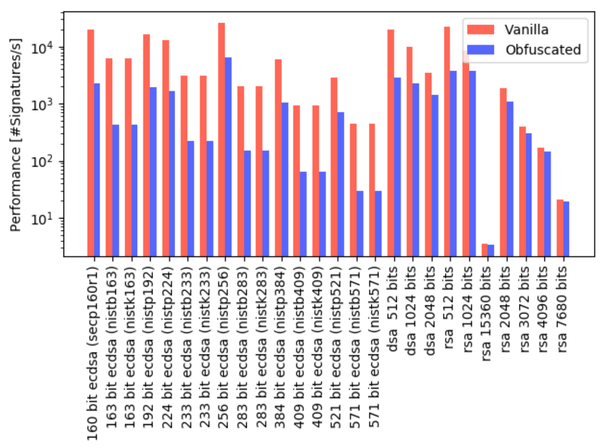


Fig. 12: OpenSSL 1.0.0f vanilla vs obfuscated asymmetric signature benchmark results. The vertical axis is scaled logarithmically and shows the throughput in the number of signatures per second. The horizontal axis denotes the different algorithms.

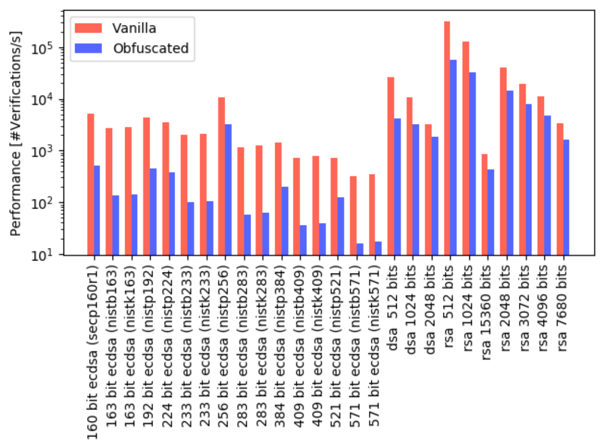


Fig. 13: OpenSSL 1.0.0f vanilla vs obfuscated asymmetric verification benchmark results. The vertical axis is scaled logarithmically and shows the throughput in the number of verifications per second. The horizontal axis denotes the different algorithms.

can be explained by the fact that the obfuscation has little performance impact on the symmetric algorithms due to optimized source code and the compiler performing loop unrolling. These optimisations result in larger and more complex blocks that are not affected by our obfuscator.

Fig. 12 and Fig. 13 show the benchmarks results for asymmetric signature generation and verification for both the vanilla and obfuscate versions. In this case, the obfuscation has a higher performance impact on the asymmetric algorithms when compared to the symmetric algorithms. This difference is explained by the different structure of the control flow graphs of both algorithm types. The asymmetric algorithms feature a larger variety of small basic blocks and tighter loops that are affected more by our obfuscation approach. The symmetric algorithms feature larger basic blocks and less loops that can additionally be unrolled by the compiler in case they are executed a constant number of times.

Fig. 14 shows the comparison of the benchmark results for asymmetric key exchange algorithms. The results feature the same performance behaviour as for the signature generation and verification as the key exchange algorithms are based on the same cryptographic primitives.

Mind that in Fig. 10, Fig. 12, Fig. 13 and Fig. 14 the throughput and performance scales are logarithmic.

The overall performance of the obfuscated asymmetric algorithms compared to the vanilla versions is $\mathbf{20.9\% \pm 23.1\%}$.

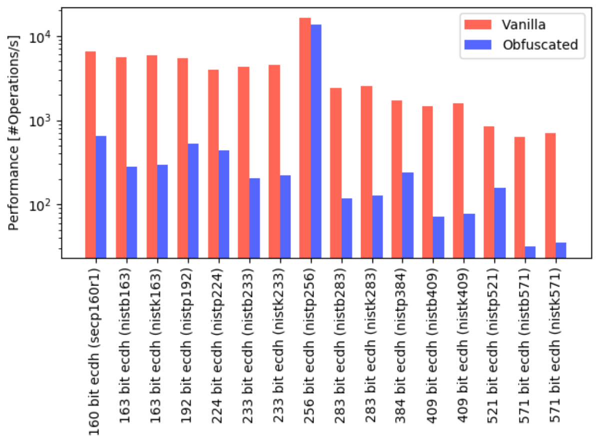


Fig. 14: OpenSSL 1.0.0f vanilla vs obfuscated asymmetric key exchange benchmark results. The vertical axis is scaled logarithmically and shows the throughput in the number of key exchanges per second. The horizontal axis denotes the different algorithms.

It heavily varies for the individual algorithms as can be see from Fig. 12, Fig. 13 and Fig. 14. This can be explained by taking the specific predicate usage of the different algorithms into account. Specifically algorithms featuring tight loops and lots of small basic blocks show a worse performance compared to algorithms with a less complicated structure. This explains why asymmetric algorithms relying on handling of large integers perform worse than symmetric encryption algorithms.

To further evaluate our results, we have analysed the [3]published benchmark results for OpenSSL 1.0.1e obfuscated with *Obfuscator-LLVM*. We found that the performance of the obfuscated asymmetric algorithms compared to the vanilla versions is $\mathbf{1.3}\% \pm \mathbf{0.7}\%$ for a worst case obfuscation. This amounts to an approximately tenfold increase in execution performance of our obfuscator compared to *Obfuscator-LLVM*.

Finally, we also compared the file sizes of the vanilla and the obfuscated libraries (*libcrypto.so* and *libssl.so*) and found out that the obfuscated file size is approximately twice as big as their vanilla counterparts: 2726 kB and 490 kB for the vanilla files versus 5412 kB and 1116 kB for the obfuscated files.

B. mbed TLS 2.5.1

Similar to OpenSSL, mbed TLS is also an open source collection of routines implementing the TLS and SSL protocols. It consists of roughly 155000 LoC and is targetting embedded and fully fledged systems likewise. One of the major uses of mbed TLS is to secure and encrypt the network communication between network clients and servers, but it also provides generic access to various cryptographic algorithms. These algorithms range from symmetric and asymmetric encryption over secure hashing to large integer arithmetic.

[3]https://github.com/obfuscator-llvm/obfuscator/wiki/Benchmarks

TABLE IX.2: Vanilla OpenSSL vs. obfuscated OpenSSL self-test results. The vanilla reference version passes all 60 tests. The obfuscated version also passes all 60 tests. Note that the time required for the obfuscated version to finish all tests is roughly 8 times the time it took the vanilla version.

mbed TLS Type	# Tests passed (of 60)	Running time
Vanilla	60	14s
Obfuscated	60	117s

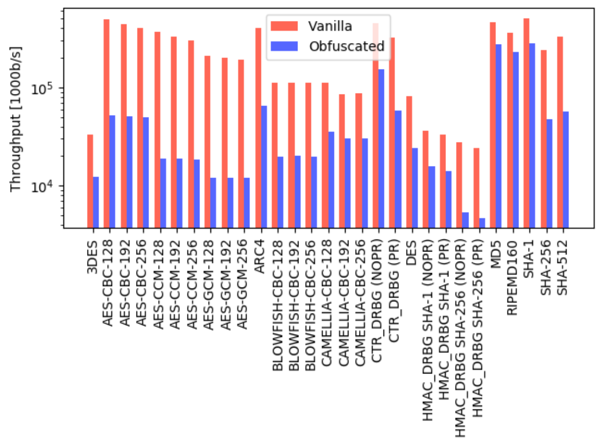


Fig. 15: mbed TLS 2.5.1 vanilla vs obfuscated symmetric algorithms benchmark results. The vertical axis is scaled logarithmically and shows the throughput in 1000 bytes per second. The horizontal axis denotes the different algorithms.

As for OpenSSL, we have found that a large part of the predicates can be dressed. Out of 21038 integer comparisons, a total of 17062 them were constant or variable point comparisons and have been dressed as infeasible predicates. This amounts to 81.1% of predicates that the obfuscator was able to dress. A total of 52437 infeasible constant predicates were inserted.

After the code was obfuscated, we executed the tests included with the library for both the vanilla and obfuscated versions. From TABLE IX.2, we can see that the obfuscated version of mbed TLS passes all self-tests in roughly 8 times the amount of time that the vanilla version requires.

In Fig. 15, we can see the benchmark results for the vanilla and obfuscated symmetric cryptography algorithms in logarithmic scale. When compared with Fig. 10, we see that this implementation of symmetric algorithms is more affected by the obfuscation. This is because the mbed TLS 2.5.1 implementation features less optimized algorithms than OpenSSL 1.0.0f counterparts. In the mbed TLS code, the obfuscation affects a larger number of small basic blocks that appear in tight unoptimized loops leading to a larger performance impact. As a result, for the symmetric algorithms the obfuscated versions reach a mean performance of $\mathbf{24.1}\% \pm \mathbf{16.5}\%$ of the vanilla versions.

In Fig. 16, we compare the benchmark values for the mbed

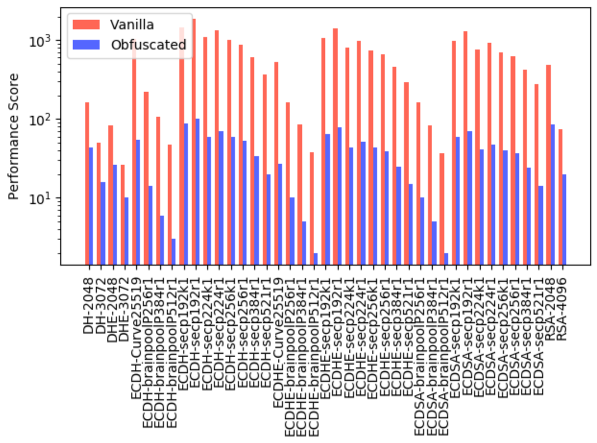


Fig. 16: mbed TLS 2.5.1 vanilla vs obfuscated asymmetric algorithms benchmark results. The vertical axis is scaled logarithmically and shows the throughput in the number of asymmetric operations per second. The horizontal axis denotes the different algorithms.

TABLE IX.3: Vanilla mbed TLS vs. obfuscated mbed TLS benchmark results. We can see the performance of the obfuscated cryptographic algorithms as percentage of the vanilla versions for different obfuscation levels.

Dressed	Inserted	Symmetric perf.	Asymmetric perf.
100%	100%	$\mathbf{24.1\% \pm 16.5\%}$	$\mathbf{9\% \pm 8.5\%}$
100%	50%	$\mathbf{34.7\% \pm 20.8\%}$	$\mathbf{13.4\% \pm 12\%}$
50%	100%	$\mathbf{25.7\% \pm 18.9\%}$	$\mathbf{8.8\% \pm 7.3\%}$
50%	50%	$\mathbf{41.1\% \pm 23\%}$	$\mathbf{14.8\% \pm 12.1\%}$
33%	33%	$\mathbf{45.6\% \pm 26.7\%}$	$\mathbf{22.2\% \pm 12.8\%}$

TLS asymmetric cryptographic algorithms in logarithm scale. In the asymmetric case the obfuscated versions reach a mean performance of $\mathbf{9\% \pm 8.5\%}$ of the vanilla versions.

We have conducted the same experiment for different combinations of dressing and constant predicate insertion percentages. TABLE IX.3 contains a complete list of the results we have found from running the benchmark. Our results show that inserting constant predicates has a larger impact on the performance compared to dressing already existing predicates. In the case where we reduced the number of dressed and inserted predicates by half, we gained almost double the performance compared to the worst case obfuscation. This shows that controlling the number of dressed and especially the number of inserted predicates allows for a fine-tuning of the performance overhead. Of course, tuning the obfuscation parameters strongly depends on the structure of the code that is obfuscated.

For the vanilla build of mbed TLS, we found the file sizes of *libmbedcrypto.a*, *libmbedtls.a* and *libmbedx509.a* to be 628 kB, 238 kB and 98 kB, respectively. When obfuscated, these libraries file sizes went up to 1367 MB, 586 kB and 301 kB, respectively. This amounts to the obfuscated files again ending up roughly twice to trice as big as their vanilla counterparts.

C. *Analysis*

We stress that the benchmark results presented above represent the worst-case scenarios where all the dressable predicates are modified and for each edge a constant predicate is added. From these results we can see that obfuscating tight loops and small blocks that are called in rapid succession (featured by large integer arithmetic) exhibit most of the penalty. The larger number of CFG edges is affected more severely by the addition of constant predicates. However, in the case of the symmetric encryption algorithms in OpenSSL with larger CFG blocks the performance of the obfuscated code is better and in some cases show almost no overhead.

To counteract heavy overhead, the obfuscator behaviour could be modified. This is possible by either leveraging LLVM's analysis logic to detect poorly obfuscable structures or by explicit user interaction. We could for example omit inserting constant predicates in these tight loops or choose not to dress the loop header predicate. Any omissions come at a price though - they weaken the obfuscation scheme. This is not surprising as less constant predicates respectively less dressed predicates imply easier reverse engineering and recovery of the original program.

X. Conclusion

We have seen how to construct infeasible constant predicates for which it is hard to prove constancy. Furthermore, we have given ways to dress existing predicates of a special form in a program to be indistinguishable from our constructed infeasible constant predicates. For now the possible forms are constant and variable point comparisons. We have shown that dressing is not possible for ordering predicates. One open question is how well the idea of dressing predicates can be extended to other types of predicates.

We have surveyed attacks on the proposed schemes by means of static and dynamic program analysis. Additionally, we have shown techniques to harden the obfuscation scheme against these attacks by adding superfluous program paths that have data dependencies on the input variables.

An important follow-up problem is the generation of inert code that can be inserted into paths following balanced or noticeable predicates featuring the required data dependencies on the input values. We also require that the output values should depend on them in a non-trivial yet functionality-preserving way. Another interesting follow-up question is how to generate the superfluous instruction stream to appear reasonable in the sense that it might be found in a non-obfuscated program as well.

References

[1] G. Arboit, "A method for watermarking java programs via opaque predicates," in *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002, pp. 102–110.

[2] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, "Opaque predicates detection by abstract interpretation," in *International Conference on Algebraic Methodology and Software Technology*. Springer, 2006, pp. 81–95.

[3] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research*, vol. 6, no. 2, pp. 155–171, 2006.

[4] A. Majumdar and C. Thomborson, "Manufacturing opaque predicates in distributed systems for code obfuscation," in *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*. Australian Computer Society, Inc., 2006, pp. 187–196.

[5] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 757–768.

[6] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Reverse Engineering, 12th Working Conference on*. IEEE, 2005, pp. 10–pp.

[7] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 674–691.

[8] M. Madou, L. Van Put, and K. De Bosschere, "Understanding obfuscated code," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 2006, pp. 268–274.

[9] J.-Y. Marion and D. Reynaud, "Dynamic binary instrumentation for deobfuscation and unpacking," in *Depth Security Conference*, 2009.

[10] Y. Guillot and A. Gazet, "Automatic binary deobfuscation," *Journal in computer virology*, vol. 6, no. 3, pp. 261–276, 2010.

[11] F. Biondi, S. Josse, A. Legay, and T. Sirvent, "Effectiveness of synthesis in concolic deobfuscation," 2015.

[12] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 275–284.

[13] J. Salwan, M.-L. Potet, and S. Bardin, "Deobfuscation of vm based software protection," http://shell-storm.org/talks/SSTIC2017_Deobfuscation_of_VM_based_software_protection.pdf, 2017.

[14] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

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

[16] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de) obfuscation tool," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2006, pp. 140–144.

[17] D. Low, "Protecting java code via code obfuscation," *Crossroads*, vol. 4, no. 3, pp. 21–23, 1998.

[18] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel, "A taxonomy of self-modifying code for obfuscation," *Computers & Security*, vol. 30, no. 8, pp. 679–691, 2011.

[19] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Still: Exploit code detection via static taint and initialization analyses," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE, 2008, pp. 289–298.

[20] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 94–109.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 317–331.

[22] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 732–744.

[23] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 189–200.

[24] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov, "An approach to the obfuscation of control-flow of sequential computer programs," in *International Conference on Information Security*. Springer, 2001, pp. 144–155.

[25] J. Ge, S. Chaudhuri, and A. Tyagi, "Control flow based obfuscation," in *Proceedings of the 5th ACM workshop on Digital rights management*. ACM, 2005, pp. 83–92.

[26] T. Toyofuku, T. Tabata, and K. Sakurai, "Program obfuscation scheme using random numbers to complicate control flow," *Lecture notes in computer science*, vol. 3823, p. 916, 2005.

[27] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, pp. 3–19, 2009.

[28] R. Rolles, "Unpacking virtualization obfuscators," in *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[29] S. Ghosh, J. Hiser, and J. W. Davidson, "Replacement attacks against vm-protected applications," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 203–214, 2012.

[30] J. Kinder, "Towards static analysis of virtualization-obfuscated binaries," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 61–70.

[31] J. Salwan and R. Thomas, "How triton can help to reverse virtual machine based software protections," https://triton.quarkslab.com/files/csaw2016-sos-rthomas-jsalwan.pdf, 2016.

[32] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1998, pp. 184–196.

[33] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[34] H. Wee, "On obfuscating point functions," in *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. ACM, 2005, pp. 523–532.

[35] R. Canetti, G. N. Rothblum, and M. Varia, "Obfuscation of hyperplane membership." in *TCC*, vol. 5978. Springer, 2010, pp. 72–89.

[36] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *NDSS*, 2008.

[37] 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.

[38] "The llvm compiler infrastructure project," http://llvm.org/, accessed: 2017-06-19.

[39] "Fnv hash," http://www.isthe.com/chongo/tech/comp/fnv/index.html, accessed: 2017-06-19.

[40] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[41] "April 2014 web server survey," https://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html, accessed: 2017-07-28.