When Are Opaque Predicates Useful?

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. However, there are many attacks known to detect opaque predicates and remove dummy code. We survey these attacks and argue that many types of programs cannot be securely obfuscated using opaque predicates. In particular we explain that most previous works on control flow obfuscation have introduced predicates that are easily distinguished from naturally occurring predicates in code, and hence easily removed by an attacker.

We state two conditions that are necessary for a program to be suitable for control flow obfuscation. We give an integrated approach to control flow obfuscation that simultaneously obfuscates real predicates and introduces opaque predicates. The opaque predicates are indistinguishable from the obfuscated real predicates in the program. If an attacker applies the usual approaches (both static and dynamic) to identify and remove opaque predicates then they are likely to remove critical functionality and introduce errors. We have implemented our obfuscator in LLVM. We provide an analysis of the performance of the resulting obfuscated code.

1 Introduction

Program obfuscation is a general tool used to protect *intellectual property* from reverse engineering. This paper is concerned with obfuscating control flow. Opaque predicates [3, 13, 14, 29, 32, 41] 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 traditionally 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, or to add code that looks relevant to the program, even though these blocks are never executed when the program is run.

There is a large literature on breaking control flow obfuscation based on opaque predicates, and we survey some of it in Section 4. Many proposals can be broken by using static analysis [17, 28, 35, 44, 49] and/or dynamic analysis [4, 7, 20, 30, 31].

1.1 Our Contributions

We survey attacks that detect opaque predicates and remove dummy code. In particular, as we discuss in detail in Section 4.4, if the opaque predicates have no resemblance (either syntactically or semantically) to the naturally occurring predicates in a program then they can be easily removed by a pattern-matching attack. We demonstrate this attack on programs obfuscated by *Obfuscator-LLVM* [23].

Another powerful type of attack tries to identify constant predicates by evaluating them on chosen inputs or by examining execution traces. A simple observation that seems to have been ignored in the obfuscation literature is that most naturally occurring predicates are easily confirmed to be non-constant using this approach. For example, we have performed an experiment using open source code (OpenSSL and mbed TLS) to study predicates that naturally appear in programs. We found that approximately 91% (in the case of OpenSSL) and 83% (in the case of mbed TLS) of all constant comparisons were comparisons with zero. Hence, simply evaluating a predicate at 0 is sufficient to detect an opaque (always false) predicate with high accuracy.

One main contribution of our paper is to explain that opaque predicates can only be applied securely when the original program has certain features (in particular, that it contains predicates that are satisfied only for rare and hard-to-find inputs). In Section 5 we state two necessary conditions for being able to securely obfuscate the control flow of a program. These conditions often do not hold. This answers the question in the title of our paper.

Our other main contribution is to give an integrated approach to control flow obfuscation that can be applied when the conditions in Section 5 are satisfied. The main idea is to ensure that opaque predicates added to the program are *indistinguishable* from obfuscations of real predicates already in the program.

Our approach exploits the well-known and widely-used obfuscation technique to obfuscate constant comparisons using hash functions [9,12,40]. Our innovations are to also use hash functions for variable comparisons and, more importantly, to build opaque predicates from hash functions that are indistinguishable from obfuscations of real predicates. We have implemented our proposed obfuscation scheme on top of the LLVM compiler infrastructure as a generic compiler plugin. We give both theoretical and experimental evidence to support our security claim that the real and opaque predicates are hard to distinguish, and we give the result of experiments that show how the obfuscation affects the execution performance.

We briefly explain the security of our obfuscation technique, when applied under the conditions in Section 5, with respect to the measures for obfuscation techniques proposed by [12,13,14,15]:

- *Potency* — We increase the control flow complexity by adding opaque predicates and bogus basic blocks. This approach to obfuscation has been widely used.
- *Resilience* — This follows immediately, since we prove that our opaque predicates are indistinguishable from obfuscations of already occurring predicates, cf. Section 6.
- *Stealth* — We do not disguise the fact that an obfuscator has been applied to the program.
- *Cost* — An analysis of the cost (with respect to runtime performance) of inserting our new type of opaque predicates is presented in Section 8.

Recall that *potency* and *resilience* mean resistance to reverse engineering by human and automated attacks respectively, while *stealth* means hiding from an attacker which transformation was applied where in the code.

A software developer may be reluctant to use an obfuscator that introduces "dummy code" into a program: How can they be certain that the obfuscator has not inserted malicious code into the program? An additional original contribution of our work is a method for the obfuscator to give evidence to the software developer or a third-party auditor that the obfuscated program is safe. Thus we identify the following new security property for obfuscation:

- *Verifiability* — A mechanism for an obfuscation tool to prove to a software developer that the obfuscated version of their program does not contain malicious code. We discuss our verifiable mechanism in Section 6.7.

2 Related Work

In this section we survey some relevant background ideas on obfuscation of control flow. Due to lack of space we cannot survey the entire topic.

The principal goal is to hide a program's control flow graph (CFG) [11]. Approaches include introducing superfluous control flow [19,34,43]. In general, a control flow transformation must ensure that the original control flow is contained in the newly generated CFG while introducing artificial complexity.

Control flow obfuscators often introduce superfluous or inert instructions interleaved with the original ones. Special care has to be taken so 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 [38, 39, 44, 45]. 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 [4,50]. Certain obfuscation schemes are weak against symbolic execution. To counteract this, control flow loops are transformed in a way that increases symbolic execution time.

Once an opaque predicate is inserted in a program, it is possible to further protect the code using transformations meant to mask the opaque predicate itself. The de-obfuscation of these additional transformations and the opaque predicates detection are problems that can be studied independently.

3 Notation and Definitions

3.1 Classes of Predicates

Here we give definitions of certain classes of predicates commonly found in programs. Let X be a finite set. A *predicate* on X is a function $P : X \to \{0, 1\}$. The element 1 will usually represent the Boolean *true* and the element 0 the Boolean *false*. A predicate is *constant* if it is the constant function.

We call a predicate $P(x) =$ "$x == c$", where c is a fixed value, a *constant comparison*. We call a predicate $P(x, y) =$ "$ax + b == y$", where a, b are fixed and x, y variables, a *variable comparison* or *variable point comparison*.

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 theory we can only consider obfuscation of a class 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 [6, 10, 47].

Definition 2. *A predicate P on X 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*

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

where $|X| = 2^n$.

Noticeable predicates can be efficiently distinguished from constant predicates by computing them on random inputs (see Section 4.3).

3.2 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 [26]. 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* $E \subseteq V \times V$ 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 location 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 $y \in \{0, 1\}$.

4 Attacks on Opaque Constant Predicates

We now survey techniques to detect obfuscated constant predicates. The possible methods involve human interaction as well as automated algorithmic interaction [17, 31].

Let $P : X \to \{0, 1\}$ be a predicate, where X is the predicate's domain. We wish to automate determining whether $P(x)$ is constant or not. If we can solve this problem, we can build an automated reverse engineering 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 an adversary can try to recover the original version of the program or a close version of it.

Sections 4.1, 4.2, 4.3, 4.4 and 4.5 give static attacks, while Sections 4.6, 4.7 and 4.8 are more dynamic in nature.

4.1 Brute Force Search

If X is a small enough set then one can efficiently execute the predicate $P(x)$ for all $|X|$ possible values $x \in X$ and so determine 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. This topic is studied in [17, Section 4]. 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.

4.2 Evaluate at Zero

Our analysis shows that typical programs exhibit a large number of constant comparisons. OpenSSL for example has 36855 integer comparisons of which 28890 are with constants. Approximately 91% of these are comparing with zero. Similary, for mbed TLS, 17062 of 21038 integer comparisons are with constants. Roughly 83% of these are comparing with zero. Hence a simple strategy to distinguish a real predicate from an opaque constant (always false) predicate is to evaluate it at zero. If the predicate is true for 0 then it is not a constant false predicate; if the predicate is false for 0 then it can be flagged for further analysis. The power of this technique seems to not have been noticed by obfuscation researchers.

4.3 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. This type of attack is studied in detail in [17, Section 3]. They call a "false negative" the situation where a probabilistic or dynamic (cf. Section 4.7) attack incorrectly classifies a non-constant predicate as being constant. They conducted an experimental study using real programs and found false negative rates of between 20 and 40 percent. In other words, running predicates in real-world programs on random inputs is not a reliable method to determine if a predicate is constant, and the removal of code blocks based on such an attack is not safe (it may destroy functionality based on loop conditions or error handling). But this check is sensible as a pre-processing step before applying more sophisticated methods to determine if a predicate is constant or not.

Table 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 $\mathbb{Z}/2^n\mathbb{Z}$.

$7y^2 - 1 \neq x^2$
$2 \mid x(x+1)$
$3 \mid x(x+1)(x+2)$
$x^2 > 0$
$7x^2 + 1 \not\equiv 0 \mod 7$
$x^2 + x + 7 \not\equiv 0 \mod 81$
$x > 0$ for $x \in I$ random where $I \subset \mathbb{N} = \mathbb{Z}_{>0}$ is a random interval

4.4 Pattern Matching

We have surveyed the literature [3, 14, 23, 32], as well as studied samples of code produced by both free and commercial obfuscation solutions, to collect examples of constant predicates. Surprisingly, relatively few predicates have been proposed, and they are used over and over again. Table 1 lists the most-used constant predicates.

A fundamental problem, mentioned in [12, Section 4.4], is that these predicates may not otherwise naturally occur in code. If the introduced predicates are of a different form to the real code, then it is easy to detect and remove them purely by searching for that syntax in the code. This leads to a dictionary attack [3, 12, 32], where one takes obfuscated predicates from the program being attacked and pattern-matches the source code against example code for the predicates in Table 1.

As an example, we consider the open source implementation of *Obfuscator-LLVM* [23]. It uses a unique static constant predicate $P(x, y) = (y < 10) \vee (2 \mid x(x+1))$ 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.

Of course, it is easy to create additional constant predicates that are not listed in Table 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 [3] 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 method 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.

We believe that pattern matching attacks have been neglected in the obfuscation literature, and that they can be used to attack many recent obfuscation schemes that introduce predicates in a "non-organic" way. For example, [34] introduces non-determinism and complexity of control flow by using particular branch instructions based on a random number generator. These predicates are of a very special form and are not likely to resemble the predicates in the original program. If an attacker knows that this "probfuscation" approach has been used then it is plausible that these branches can be removed using a pattern-matching attack.

As another example, Palsberg et al. [33] and Xu et al. [48] describe dynamic opaque predicates of a very special form that rely on certain correlated variables. Once again, if an attacker knows that this approach has been used then it is plausible that a pattern-matching approach can identify these predicates and therefore simplify the control flow graph. Similar remarks apply to opaque predicates based on aliasing [14] or 3SAT [41]. In short, *predicates added to the program have to resemble the syntax of naturally occurring program code.*

4.5 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

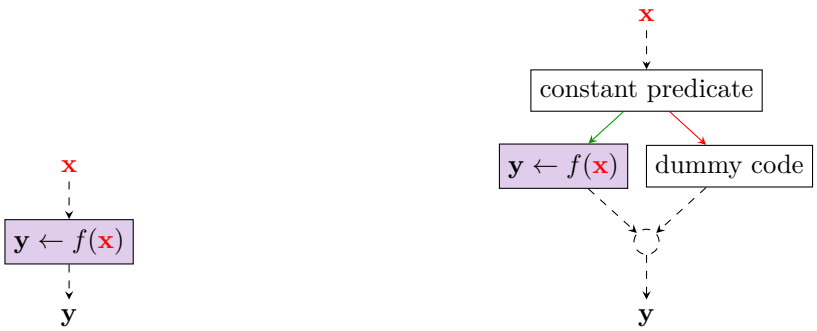

Figure 1: Example of extracting the original CFG from obfuscated CFG using taint analysis. The nodes that have no data dependence on the input can be ignored when extracting the logic that operates on the input. In (a) the input basic block produces an output $\mathbf{y}$ that depends on the input $\mathbf{x}$. Note that the inserted basic block in (b) does not depend on the input $\mathbf{x}$.

size of the space X and the *complexity* of P. *Symbolic execution* backed by SMT solvers is possible in a static and a dynamic context [8, 37, 42].

4.6 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. Figure 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 the obfuscated function. Taint analysis [38, 45] 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.

The problem of identifying dummy code has been studied by many authors. In the context of compiler construction, optimization and removal of dead code are important topics [21, 22, 24, 25, 36]. A recent work by Wang et al. [46] distinguishes "irrelevant code" (we call it "inert") that can be executed, but has no effect on the result of the program, and "unreachable code" that is never executed.

4.7 Execution Traces

This is essentially a dynamic version of the probabilistic check mentioned in Section 4.3. One executes the obfuscated program in a 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 correctly identify nearly-constant predicates such as loop terminations.

This approach allows to determine that predicates are real predicates from the original program, however one cannot immediately conclude that program blocks that have not been executed are dummy code. Removing such predicates from a program can have a drastic effect on the reliability and correctness of the program. Dalla Preda et al. [17] write that such attacks incorrectly classify many predicates as opaque when they are not. They write "dynamic attacks do not provide a trustful solution".

4.8 Mutation Attack

A way to determine if a block of instructions has an influence on the execution of a program is to introduce a mutation into those instructions (e.g. replace some instructions with something different) and execute the program. If the program crashes then we conclude that this code is critical to the program. If the program still executes correctly on a range of inputs then one might conclude that the block is dummy code that can be removed. While this is potentially a powerful attack in some settings, it is hard to automate as it requires to determine whether a program is behaving "correctly". The complexity of the attack also scales badly as the number of different inputs needed to test the

program may be exponential. It is well-known that software testing is a hard problem, which suggests that this de-obfuscation strategy is of limited value in practice.

5 When Are Opaque Predicates Useful?

Our experience is that it only makes sense to use opaque predicates as an obfuscation tool under the following two conditions:

1. The program being obfuscated has a lot of conditional branches, including many constant comparisons ($x == c$) with "random" constants c, or variable comparisons ($ax+b == y$), again with "random" constants a, b. We will call this *rich control flow*.

2. There is an algorithm to generate dummy code blocks whose instructions and data dependencies are indistinguishable from real code blocks in the program.

The first condition is needed to avoid the attacks mentioned in Section 4.2 and Section 4.3. The second condition is needed to prevent the attacks in Section 4.6.

Both these conditions depend on the program (or class of programs) being obfuscated. The second condition is implicit in all previous work on control flow obfuscation (and is made explicit in [34]). There are various proposals in the literature, for example Banescu and Pretschner [5] write that "dead code can be a buggy version of the other branch".

There are types of programs for which these two conditions do not hold. This statement is obvious, but does not seem to have been clearly stated in the literature on obfuscation. The implication is that control flow obfuscation using opaque predicates is not appropriate for some programs. We argue that, for such programs, control flow is probably not an important asset and so the software developer would probably not be interested in protecting it using a control flow obfuscator. Instead, the assets in the program may be secret data or other properties, in which case they should be protected using an appropriate obfuscation tool.

6 Obfuscating Predicates

For the remainder of the paper we restrict to situations when the two conditions in Section 5 hold. We will give an obfuscation tool that avoids the powerful attacks we discussed in Section 4.4 and Section 4.5.

The main idea is to simultaneously obfuscate (we call it *dress*) the existing predicates in the program and also introduce opaque predicates. The obfuscated real predicates are syntactically identical to the opaque predicates, and hence the real predicates and the opaque predicates are indistinguishable.

6.1 Obfuscating constant comparisons

It is standard [16, 27] to 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 proposed to hide code checking for malware triggers [40] for example.

Let H be a cryptographic hash function. Let $\mathcal{C}$ be the class consisting of all constant comparison predicates $P(x) =$ "$x == c$" where x has k bits. (A more intelligent tool would obfuscate only those constant comparison predicates that are not easily reverse-engineered using the methods in Sections 4.2 and 4.3.) To obfuscate a comparison predicate "$x == c$" for some k-bit constant c, the obfuscator chooses a hash function H with n-bit output (where $n > k$), chooses a random $t \in \{0,1\}^{n-k}$ and computes $h = H(c\|t)$. The obfuscated predicate $P(x)$, which is specified by the pair (t, h), computes $y = H(x\|t)$ and then checks if $y = h$.

It is clear that the obfuscated predicate outputs true on the correct input $x = c$. With overwhelming probability it outputs false on all other inputs.

Looking ahead we now explain how we will use the same construction to generate an opaque predicate (i.e., an obfuscation of the constant function): we choose a random $h \in \{0,1\}^n$ and $t \in \{0,1\}^{n-k}$ and publish the obfuscated predicate $P(x)$ that computes $y = H(x\|t)$ and checks if $y = h$. With high probability the predicate is always false since there would be no k-bit value x that satisfies the equation when $n > k$.

Note that it is possible to obfuscate variable comparison predicates in a similar way.

The following lemma and theorem are the basic tools 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 concatenation.

Lemma 1. *Let $H : \{0,1\}^n \to \{0,1\}^n$ be a cryptographic hash function. For each $1 \le k \le n$ 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 $x = t_n$. □

Theorem 1. *Let $X = \{0,1\}^k \subseteq \{0,1\}^n$ and let $H : \{0,1\}^n \to \{0,1\}^n$ be a preimage-resistant hash function. Let $\mathcal{C}$ be the class of predicates $P(x) =$ "$x == c$" where c is uniformly sampled from X. Then there does not exist any efficient adversary (i.e., running time polynomial in k) that, for any obfuscated predicate from $\mathcal{C}$, 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-resistant. □

6.2 Obfuscating Variable Comparisons using Hash functions

We can obfuscate variable point comparisons $P(x, y) =$ "$ax + b == y$" using hash functions. Here a and b are constants and $x, y \in \mathbb{Z}$ are the variables. Our solution will hide b, which is enough to make it hard to find a solution (x, y) to the predicate.

Suppose that x and y are at most k-bits in length and let $X = [0, 2^k)$ be the set of k-bit integers. So $x, y \in X$. Let $n > k$ and let $H : \{0,1\}^n \to \{0,1\}^n$ be a preimage-resistant hash function. The obfuscator chooses a random $t \in \{0,1\}^{n-k}$ and a random $r \in X$, sets $h = H(r\|t)$ and $u = (r + b)$ and publishes a, u, h. On input x, y the obfuscated program computes $H(ax + u - y\|t)$ and then checks if this equals h. It is clear that the obfuscated program is correct: any input (x, y) for which P is true will also evaluate to true. The probability of a false positive can be made negligible by taking n large enough (e.g., $n = 3k$).

Note that h hides the value r and so u hides the real value b.

Looking ahead, we now explain how to make an opaque predicate of the same form. We choose a random $t \in \{0,1\}^{n-k}$, random $a, u \in X$, and a random $h \in \{0,1\}^n$ and publish the obfuscated predicate $P(x, y)$ that checks whether $h = H(ax + u - y\|t)$. With high probability there is no input (x, y) for which this predicate evaluates to true. Hence we have a constant (false) predicate that is indistinguishable from the obfuscated real predicate.

6.3 Full obfuscation tool

Our tool first dresses (obfuscates) all point-comparison and variable-comparison predicates in the original program. The second step is to introduce superfluous control flow, by using constant predicates. The core of our idea is to obfuscate the constant predicates (make them opaque) so that they are indistinguishable from the dressing of the original predicates in the program. To make the control flow more complex, we take a full or partial CFG and prepend it with a constant predicate. The inserted CFG can be of arbitrary complexity as it will never be executed. The situation is depicted in Figure 2.

This way of introducing opaque predicates avoids the pattern-matching attack from Section 4.4: an attacker cannot remove all predicates that "look like" opaque predicates, as some of them are real comparisons and their removal will not maintain the correctness of the program.

6.4 Theoretical Description

Our tool first dresses all point-comparison and variable-comparison predicates in the original program, using the techniques presented in the previous sections. The second step is to introduce superfluous

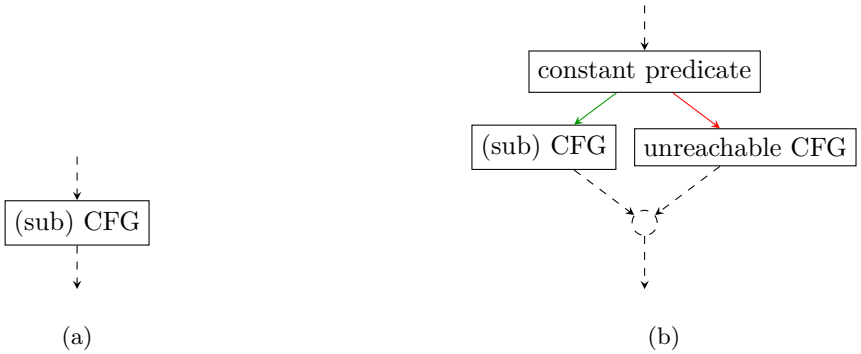

Figure 2: Obfuscating control flow graph using a constant predicate. The input graph depicted in (a) is prepended by a constant predicate and random superfluous code is inserted in the branch that is never taken. This produces the output graph depicted in (b).

control flow, by using constant predicates. The core of our idea is to obfuscate the constant predicates (make them opaque) so that they are indistinguishable from the dressing of the original predicates in the program.

To make the control flow more complex, we take a full or partial CFG and prepend it with a constant predicate. An example for a program with a single basic block 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 by B_t and the entry block of the inserted inert CFG is given by B_o.

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.

We now detail the first stage of the obfuscator: to obfuscate all point comparisons and variable comparisons in the original program. Consider a constant point comparison $P(x) =$"$x == c$". Our obfuscation scheme $\mathcal{O}$ transforms this predicate according to

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

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

We now explain the analogous process for variable point comparisons, following the construction of Section 6.2. Suppose a variable point comparison $P(x, y) =$"$x == y$". To dress it, we generate random integers r, t and compute the hash $h_r = H(r\|t)$. The comparison predicate is the obfuscated (dressed) as follows.

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

In the second stage, we introduce opaque predicates that have a similar structure as the above types. This way of introducing opaque predicates avoids the pattern-matching attack from Section 4.4: an attacker cannot remove all predicates that "look like" opaque predicates, as some of them are real comparisons and their removal will not maintain the correctness of the program.

6.5 Adding Complexity to Control Flow using Non-Constant Predicates

One can take a code block B and replace it with a conditional branch that executes two syntactically different but semantically equivalent blocks B_0 and B_1. Pawlowski et al. [34] mention a method called

(a) (b)

Figure 3: A control flow graph is obfuscated using a balanced/noticeable predicate. In (a) the path branches to the original (sub) CFG whereas in (b) the original (sub) CFG was appended to or interleaved with the inert CFG.

"transform basic blocks" to produce semantically equivalent code blocks. This also adds spurious complexity to control flow. We picture this below.

$$\begin{bmatrix} z \leftarrow (x \geq y) \\ \mathbf{BCOND}_z(B_0, B_1) \end{bmatrix}$$

Once again, the main idea is to ensure that the predicate in the conditional branch is chosen to have the same format and data dependencies as real predicates in the program. Hence the added predicates are "opaque" in the sense that they cannot be removed by pattern-matching methods.

Of course there may be static or dynamic methods to determine that the code blocks B_0 and B_1 have equivalent functionality, but this is out of scope of our paper.

Figure 3 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. One now has the desired objective of complicating the control flow, but with predicates that are not always constant.

6.6 Security Analysis

Our approach achieves resilience, since the introduced opaque predicates are indistinguishable from the obfuscated real predicates.

We now discuss the potency of our obfuscation approach. As explained in Sections 4.3 and 4.7, simple probabilistic and dynamic attacks are likely to incorrectly flag real predicates as opaque, and result in real code being removed by an attacker. It is true that an adversary can identify some predicates (such as loop terminations) that will be seen to be not constant. We require that the real program has sufficiently complex control flow for our solution to resist dynamic attacks.

We further claim that the other static attacks cannot distinguish between the obfuscations of the original and inserted opaque predicates. Using an SMT solver (see Section 4.5) will not help an adversary to identify the constant predicates, assuming the hash function is preimage resistant.

Due to our assumption, the dummy code blocks have a dependency on the input variable(s) $\mathbf{x}$ in the inert CFG in Figure 1b as well as generate an output dependency for the output variable(s) $\mathbf{y}$. This way both possible execution paths will depend on $\mathbf{x}$ and generate dependencies for $\mathbf{y}$ and so an adversary will not be able to identify the superfluous path without having to solve the predicate. Hence our approach gives protection against taint analysis, as described in Section 4.6.

Our solution is not intended to provide stealth.

6.7 Verifiable Obfuscation

A user may be concerned that the obfuscator is malicious. How can the user be sure that the code introduced by the obfuscator is not malicious code that can be triggered by some secret input known

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

Figure 4: The source code of a sample function that returns the sum of all even integers $i \in [0, 20000)$.

to the obfuscator? As such, we would like to provide a verification mechanism in our obfuscator to prove to the user that the inserted code is never executed.

The above obfuscator inserts constant predicates of the form

$$\begin{bmatrix} x \leftarrow \cdots \\ h \leftarrow \mathbf{H}(x\|t) \\ y \leftarrow \mathbf{CMP}(h, h_c) \\ \mathbf{BCOND}_y(B_0, B_1) \end{bmatrix}$$

where t and h_c are randomly chosen constants in the program, B_0 is a block of dummy code (executed only when $h = h_c$, which is never the case with overwhelming probability), and B_1 is real program code.

One approach would be to prove that the code block B_0 is not malicious, but this seems to be a hard problem (especially since B_0 might interact in a non-obvious way with other program blocks). A simpler approach is for the obfuscator to give evidence that there is not a secret input x^* such that $h(x^*\|t) = h_c$. If the user is convinced that the code block B_0 can never be executed, then it does not matter what instructions are in the code block.

We now explain how an obfuscator can prove that it is not behaving maliciously. Suppose the obfuscator constructs the random value h_c by computing $H(x'\|t')$ for random binary strings x', t' of the same bitlengths as x and t, and such that $t' \neq t$. The obfuscator can provide $x'\|t'$ to an auditor. The auditor can check that $t' \neq t$ and that $h_c = H(x'\|t')$. For a malicious obfuscator to be able to know x such that $H(x\|t) = h_c$ it is necessary for the obfuscator to find a non-trivial collision $H(x\|t) = H(x'\|t')$. If H is a collision-resistant hash function then the auditor therefore gets an assurance that the obfuscator does not know an input x such that the obfuscated program computes the dummy block B_0.

7 Implementation

We have implemented our proposed obfuscation scheme as a plug-in on top of the LLVM compiler infrastructure [26]. 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. For our implementation, we have used the 64-bit version of the *FNV hash* [2]. Though not strong enough for classical crypto applications, this seems reasonable for obfuscation applications. Of course, if needed one could use a stronger hash function, but this would lead to a greater slow-down. We have experimentally tested the robustness of this hash function using the SMT solver Z3 [18] 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.

To see how our obfuscator transforms the control flow graph of a function, we consider the simple function listed in Figure 4. The decompiler-generated output of the compiled binary is virtually identical to Figure 4. Figure 5 lists the decompiler-generated output of the obfuscated binary generated from the source code in Figure 4. Due to the dressing (obfuscation) of the existing predicates and the additional constant predicates, the decompiler was not able to generate any immediately meaningful source code. In particular, lines 7-8 in Figure 5 are an example of a constant comparison predicate as described in Section 6. Lines 21-22 in Figure 5 are one instance of a variable comparison predicate. Due to our construction, it is infeasible to tell whether they are dressed or inserted constant predicates.

```
1  __int64 foo() {
2    /* ... */
3    v14 = 0LL;
4    v13 = 0LL;
5    while ( 1 ) {
6  LABEL_2:
7      LODWORD(v0) = fnv64_u64(v13);
8      if (v0 == -617515315672706485 3LL) {
9        fnv64_u64(v13);
10       return v14;
11     }
12     LODWORD(v6) = fnv64_u64(v13);
13     if (v6 == 701472864409536690 2LL)
14       goto LABEL_18;
15     *(_QWORD *)&v1 = v13;
16     *((_QWORD *)&v1 + 1) = (unsigned __int128)v13 >> 64;
17     v12 = v1 % 2;
18     LODWORD(v2) = fnv64_u64(v12);
19     if (v2 == -628478186066737721 1LL)
20       break;
21     LODWORD(v7) = fnv64_u64(v13 - v12 + 23);
22     if (v7 != 576192885975559253 4LL)
23       goto LABEL_6;
24 LABEL_18:
25     while (1) {
26       LODWORD(v10) = fnv64_u64(-329LL);
27       if (v10 == 678449759672649689 8LL)
28         break;
29       v4 = v13++;
30       LODWORD(v11) = fnv64_u64(v4 - v13 + 111);
31       if (v11 != 187402020467397606 5LL)
32         break;
33 LABEL_6:
34       v3 = v14;
35       v14 += v13;
36       LODWORD(v9) = fnv64_u64(v3 - v13 + 7);
37       if (v9 == 900005853637713081LL)
38         goto LABEL_2;
39     }
40   }
41   LODWORD(v8) = fnv64_u64(v13 - v12 + 118);
42   if (v8 != 303587327747712972 5LL)
43     goto LABEL_18;
44   return v14;
```

Figure 5: Decompiler output of the obfuscated version of Figure 4.

Note that given the original source, we are able to match the predicate of line 7 in Figure 4 with lines 18-19 in Figure 5. However, we cannot immediately match the loop predicate of line 4 in Figure 4 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.

8 Performance Evaluation

In this section, we describe our experiments using the implementation of our obfuscator and discuss the results. The 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. 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: OpenSSL 1.1.0f and mbed TLS 2.5.1. OpenSSL was already used for testing the performance of the Obfuscator-LLVM presented in [23]. Both libraries are large software projects deployed in real-world applications and feature a reasonable variety of program constructs. Both libraries come with

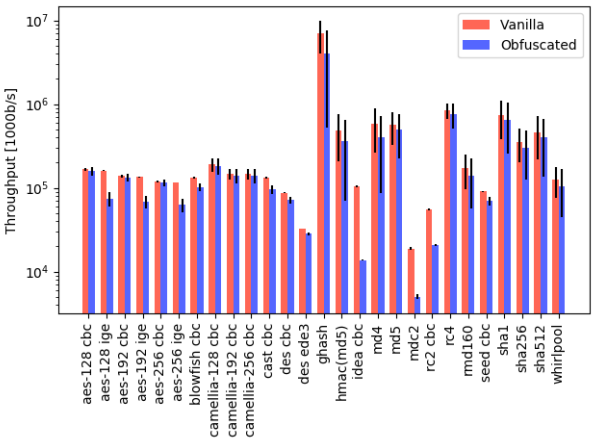

Figure 6: OpenSSL 1.0.0f vanilla vs obfuscated symmetric algorithms benchmark results.

benchmarking logic which enables us to measure the performance impact introduced by our obfuscator.

We have additionally measured the fraction of constant comparison predicates that involve a constant different from zero. In both software libraries they make up a sizeable proportion of the total number of constant comparisons.

8.1 Experiments

OpenSSL and mbed TLS are open source collections of routines implementing cryptographic protocols. They consist of roughly 470K LoC and 155K LoC, respectively, and are widely used [1].

In our analysis for OpenSSL, 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. Moreover, a total of 88458 obfuscated constant predicates were inserted. We have found that approximately 91% of all constant comparisons were comparing with zero.

For mbed TLS, we found that out of 21038 integer comparisons, a total of 17062 of them were constant or variable point comparisons. This amounts to 81.1% of predicates that the obfuscator was able to dress. A total of 52437 obfuscated constant predicates were inserted. In this case, we have found that roughly 83% of all constant comparisons were comparing with zero.

We see that OpenSSL and mbed TLS are not ideal targets for our obfuscation technique since the conditions of Section 5 are only partially met. The libraries certainly exhibit a rich control flow structure with many constant comparisons. On the other hand, more than 80% of them are comparisons with zero. Only among the comparisons with non-zero, indistinguishable predicates will work well. Nevertheless, we still believe these libraries are good testing targets to due their large codebase size and benchmarking capabilities.

Next, we executed the benchmark tests for the symmetric cryptographic algorithms for both the vanilla and obfuscated versions. We have computed the mean performance for each algorithm over all the block sizes along with the standard deviation, shown on Figure 6. As we can see, the performance of the obfuscated version is not too far from the vanilla version.

We also found that the obfuscation has a higher performance impact on 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 overall performance (running time of the original code divided by running time of obfuscated code, as a percentage) of the asymmetric algorithms is $\mathbf{20.9}\% \pm \mathbf{23.1}\%$. 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.

To further evaluate our results, we have analysed the published benchmark results[1] for OpenSSL 1.0.1e obfuscated with *Obfuscator-LLVM*. We found that the performance of the obfuscated asymmetric

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

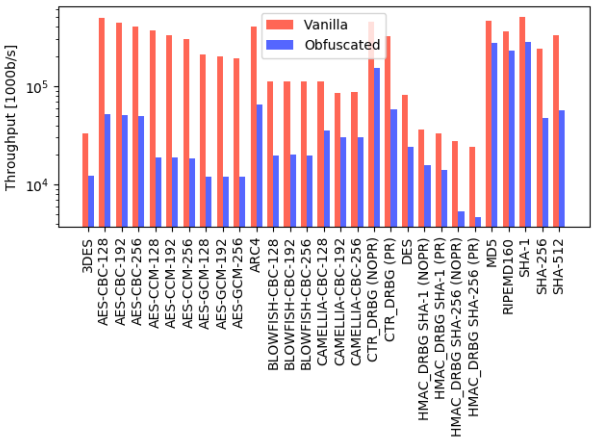

Figure 7: mbed TLS 2.5.1 vanilla vs obfuscated symmetric algorithms benchmark results.

Table 2: 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.

Dress./Ins.	Sym. perf.	Asym. 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%	$\mathbf{45.6\% \pm 26.7\%}$	$\mathbf{22.2\% \pm 12.8\%}$

algorithms compared to the vanilla versions is $\mathbf{1.3\% \pm 0.7\%}$ for a worst case obfuscation. Hence our obfuscated code is running approximately ten times faster than code generated by *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: 2726 kB and 490 kB for the vanilla files versus 5412 kB and 1116 kB for the obfuscated files.

In Figure 7, we can see the benchmark results for the vanilla and obfuscated symmetric cryptography algorithms in logarithmic scale. When compared with Figure 6, we see that this implementation of symmetric algorithms is more affected by the obfuscation. As a result, for the symmetric algorithms the obfuscated versions reach a mean performance of $\mathbf{24.1\% \pm 16.5\%}$ of the vanilla versions. We also found that the obfuscated versions of asymmetric algorithms 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 2 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. Thus controlling the number of dressed/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.

9 Conclusion

We have seen for which type of programs opaque predicates are useful. We have given ways to dress existing predicates of a special form in a program to be indistinguishable from our obfuscated constant 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 interesting follow-up question is verifiability of obfuscated programs.

Acknowledgements

We thank the Marsden Fund of the Royal Society of New Zealand for funding this research, and the reviewers for helpful comments.

References

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

[2] FNV hash. http://www.isthe.com/chongo/tech/comp/fnv/index.html, 2017. Accessed: 2017-06-19.

[3] Genevieve Arboit. A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research*, pages 102–110, 2002.

[4] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM, 2016.

[5] Sebastian Banescu and Alexander Pretschner. A tutorial on software obfuscation. In *Advances in Computers*, volume 108, pages 283–353. Elsevier, 2018.

[6] Boaz Barak, Nir Bitansky, Ran Canetti, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Obfuscation for evasive functions. In *TCC 2014*, volume 8349 of *Lecture Notes in Computer Science*, pages 26–51. Springer, 2014.

[7] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70:500–515, 2017.

[8] Cristian Cadar, Daniel Dunbar, and R. Engler Dawson. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[9] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 1997.

[10] Ran Canetti, Guy N Rothblum, and Mayank Varia. Obfuscation of hyperplane membership. In *TCC*, volume 5978, pages 72–89. Springer, 2010.

[11] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *International Conference on Information Security*, pages 144–155. Springer, 2001.

[12] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[14] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[15] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002.

[16] Giovanni Di Crescenzo, Lisa Bahler, Brian A. Coan, Yuriy Polyakov, Kurt Rohloff, and David Bruce Cousins. Practical implementations of program obfuscators for point functions. In *HPCS 2016*, pages 460–467. IEEE, 2016.

[17] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque predicates detection by abstract interpretation. In *International Conference on Algebraic Methodology and Software Technology*, pages 81–95. Springer, 2006.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[19] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control flow based obfuscation. In *Proceedings of the 5th ACM workshop on Digital rights management*, pages 83–92. ACM, 2005.

[20] Yoann Guillot and Alexandre Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[21] R. Gupta, D. A. Benson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, 1997.

[22] L. Jones, D. Christman, S. Banescu, and M. Carlisle. Bytewise: A case study in neural network obfuscation identification. In *8th Annual Computing and Communication Workshop and Conference*, pages 155–164, 2018.

[23] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9, 2015.

[24] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[25] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *SIGPLAN'94*, pages 147–158. ACM, 1994.

[26] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004.

[27] Ben Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2004.

[28] M. Madou, L. Van Put, and K. De Bosschere. Understanding obfuscated code. In *14th IEEE International Conference on Program Comprehension*, pages 268–274, 2006.

[29] Anirban Majumdar and Clark Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proceedings of the 29th Australasian Computer Science Conference*, pages 187–196. Australian Computer Society, Inc., 2006.

[30] Jean-Yves Marion and Daniel Reynaud. Dynamic binary instrumentation for deobfuscation and unpacking. In *In-Depth Security Conference*, 2009.

[31] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 757–768. ACM, 2015.

[32] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[33] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *ACSAC 2000*, pages 308–316. IEEE Computer Society, 2000.

[34] Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: An obfuscation approach using probabilistic control flows. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 165–185, 2016.

[35] R. Krishna Ram Prakash, P. P. Amritha, and M. Sethumadhavan. Opaque predicate detection by static analysis of binary executables. In *SSCC 2017*, volume 746 of *Communications in Computer and Information Science*, pages 250–258. Springer, 2017.

[36] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[37] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC*, pages 31–54, 2015.

[38] 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 *2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

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

[40] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[41] Brendan Sheridan and Micah Sherr. On manufacturing resilient opaque constructs against static analysis. In *ESORICS 2016*, pages 39–58. Springer, 2016.

[42] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*. The Internet Society, 2015.

[43] Tatsuya Toyofuku, Toshihiro Tabata, and Kouichi Sakurai. Program obfuscation scheme using random numbers to complicate control flow. In *Embedded and Ubiquitous Computing – EUC 2005 Workshops*, pages 916–925. Springer, 2005.

[44] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering*, pages 45–54, 2005.

[45] X. Wang, Y. Jhi, S. Zhu, and P. Liu. Still: Exploit code detection via static taint and initialization analyses. In *Annual Computer Security Applications Conference*, pages 289–298, 2008.

[46] Xing Wang, Yingzhou Zhang, Lian Zhao, and Xinghao Chen. Dead code detection method based on program slicing. In *CyberC 2017*, pages 155–158. IEEE, 2017.

[47] Hoeteck Wee. On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532. ACM, 2005.

[48] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *ISC 2016*, volume 9866 of *Lecture Notes in Computer Science*, pages 323–342. Springer, 2016.

[49] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691, 2015.

[50] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 732–744. ACM, 2015.