Limits of Static Analysis for Malware Detection

Andreas Moser, Christopher Kruegel, and Engin Kirda
Secure Systems Lab
Technical University Vienna
{andy,chris,ek}@seclab.tuwien.ac.at

Abstract

Malicious code is an increasingly important problem that threatens the security of computer systems. The traditional line of defense against malware is composed of malware detectors such as virus and spyware scanners. Unfortunately, both researchers and malware authors have demonstrated that these scanners, which use pattern matching to identify malware, can be easily evaded by simple code transformations. To address this shortcoming, more powerful malware detectors have been proposed. These tools rely on semantic signatures and employ static analysis techniques such as model checking and theorem proving to perform detection. While it has been shown that these systems are highly effective in identifying current malware, it is less clear how successful they would be against adversaries that take into account the novel detection mechanisms.

The goal of this paper is to explore the limits of static analysis for the detection of malicious code. To this end, we present a binary obfuscation scheme that relies on the idea of opaque constants, which are primitives that allow us to load a constant into a register such that an analysis tool cannot determine its value. Based on opaque constants, we build obfuscation transformations that obscure program control flow, disguise access to local and global variables, and interrupt tracking of values held in processor registers. Using our proposed obfuscation approach, we were able to show that advanced semantics-based malware detectors can be evaded. Moreover, our opaque constant primitive can be applied in a way such that is provably hard to analyze for any static code analyzer. This demonstrates that static analysis techniques alone might no longer be sufficient to identify malware.

1 Introduction

Malicious code (or malware) is defined as software that fulfills the harmful intent of an attacker. The damage caused by malware has dramatically increased in the past few years [8]. One reason is the rising popularity of the Internet and the resulting increase in the number of available vulnerable machines because of security-unaware users. Another reason is the elevated sophistication of the malicious code itself.

Current systems to detect malicious code (most prominently, virus scanners) are largely based on syntactic signatures. That is, these systems are equipped with a database of regular expressions that specify byte or instruction sequences that are considered malicious. A program is declared malware when one of the signatures is identified in the program's code.

Recent work [2] has demonstrated that techniques such as *polymorphism* and *metamorphism* are successful in evading commercial virus scanners. The reason is that syntactic signatures are ignorant of the semantics of instructions. To address this problem, a novel class of *semantics-aware* malware detectors was proposed. These detectors [3, 10, 11] operate with abstract models, or templates, that describe the behavior of malicious code. Because the syntactic properties of code are (largely) ignored, these techniques are (mostly) resilient against the evasion attempts discussed above. The premise of semantics-aware malware detectors is that semantic properties are more difficult to morph in an automated fashion than syntactic properties. While this is most likely true, the extent to which this is more difficult is less obvious. On one hand, semantics-aware detection faces the challenge that the problem of deciding whether a certain piece of code exhibits a certain behavior is undecidable in the general case. On the other hand, it is also not trivial for an attacker to automatically generate semantically equivalent code.

The question that we address in this paper is the following: *How difficult is it for an attacker to evade semantics-based malware detectors that use powerful static analysis to identify malicious code?* We try to answer this question by introducing a binary code obfuscation technique that makes it difficult for an advanced, semantics-based malware detector to properly determine the effect of a piece of code. For this obfuscation process, we use a primitive known as *opaque constant*, which denotes a code sequence to load a constant into a processor register whose value cannot be determined statically. Based on opaque constants, we build a number of obfuscation transformations that are difficult to analyze statically.

Given our obfuscation scheme, the next question that needs to be addressed is how these transformations should be applied to a program. The easiest way, and the approach chosen by most previous obfuscation approaches [6, 20], is to work on the program's source code. Applying obfuscation at the source code level is the normal choice when the distributor of a binary controls the source (e.g., to protect intellectual property). For malware that is spreading in the wild, source code is typically not available. Also, malware authors are often reluctant to revealing their source code to make analysis more difficult. Thus, to guard against objections that our presented threats are unrealistic, we present a solution that operates directly on binaries.

The core contributions of our paper are as follows:

- We present a binary obfuscation scheme based on the idea of opaque constants. This scheme allows us to demonstrate that static analysis of advanced malware detectors can be thwarted by scrambling control flow and hiding data locations and usage.
- We introduce a binary rewriting tool that allows us to obfuscate Windows and Linux binary programs for which no source code or debug information is available.
- We present experimental results that demonstrate that semantics-aware malware detectors can be evaded successfully. In addition, we show that our binary transformations are robust, allowing us to run real-world obfuscated binaries under both Linux and Windows.

The code obfuscation scheme introduced in this paper provides a strong indication that static analysis alone might not be sufficient to detect malicious code. In particular, we introduce an obfuscation scheme that is provably hard to analyze statically. Because of the many ways in which code can be obfuscated and the fundamental limits in what can be decided statically, we firmly believe that dynamic analysis is a necessary complement to static detection techniques. The reason is that dynamic techniques can monitor the instructions that are actually executed by a program and thus, are immune to many code obfuscating transformations.

2 Code Obfuscation

In this section, we present the concepts of the transformations that we apply to make the code of a binary difficult to analyze statically. As with most obfuscation approaches, the basic idea behind our transformations is that either some instructions of the original code are replaced by program fragments that are semantically equivalent but more difficult to analyze, or that additional instructions are added to the program that do not change its behavior.

2.1 Opaque Constants

Constant values are ubiquitous in binary code, be it as the target of a control flow instruction, the address of a variable, or an immediate operand of an arithmetic instruction. In its simplest form, a constant is loaded into a register (expressed by a `move constant, $register` instruction). An important obfuscation technique that we present in this paper is based on the idea of replacing this load operation with a set of semantically equivalent instructions that are difficult to analyze statically. That is, we generate a code sequence that always produces the same result (i.e., a given constant), although this fact would be difficult to detect from static analysis.

```
int zero[32] = { z_31, z_30, ... , z_0 };
int one[32]  = { o_31, o_30, ... , o_0 };

int unknown = load_from_random_address();
int constant = 0;

for (i = 0; i < 32; ++i) {
  if (bit_at_position(unknown, i) == 0)
    constant = constant xor zero[i];
  else
    constant = constant xor one[i];
}

constant = constant or  set_ones;
constant = constant and set_zeros;
```

Figure 1. Opaque constant calculation

Simple Opaque Constant Calculation Figure 1 shows one approach to create a code sequence that makes use of random input and different intermediate variable values on different branches. In this code sequence, the value `unknown` is a random value loaded during runtime. To prepare the opaque constant calculation, the bits of the constant that we aim to create have to be randomly partitioned into two groups. The values of the arrays `zero` and `one` are crafted such that after the `for` loop, all bits of the first group have the correct, final value, while those of the second group depend on the random input (and thus, are unknown). Then, using the appropriate values for `set_ones` and `set_zeros`, all bits of the second group are forced to

```
boolean $v_1$, ..., $v_m$, $\overline{v_1}$, ..., $\overline{v_m}$;

boolean *$V_{11}$, *$V_{12}$, *$V_{13}$;
...
boolean *$V_{n1}$, *$V_{n2}$, *$V_{n3}$;

constant = 1;
for (i = 0; i < n; ++i)
  if !(*$V_{i1}$) && !(*$V_{i2}$) && !(*$V_{i3}$)
    constant = 0;
```

Figure 2. Opaque constant based on 3SAT

their correct values (while those of the first group are left unchanged). The result is that all bits of `constant` hold the desired value at the end of the execution of the code.

An important question is how the arrays `zero` and `one` can be prepared such that all bits of the first group are guaranteed to hold their correct value. This can be accomplished by ensuring that, for each i, all bits that belong to the first group have the same value for the two array elements `zero[i]` and `one[i]`. Thus, independent of whether `zero[i]` or `one[i]` is used in the `xor` operation with `constant`, the values of all bits in the first group are known after each loop iteration. Of course, the bits that belong to the second group can be randomly chosen for all elements `zero[i]` and `one[i]`. Thus, the value of `constant` itself is different after each loop iteration. Because a static analyzer cannot determine the exact path that will be chosen during execution, the number of possible constant values doubles after each loop iteration. In such a case, the static analyzer would likely have to resort to approximation, in which case the exact knowledge of the constant is lost.

This problem could be addressed for example by introducing a more complex encoding for the constant. If we use for instance the relationship between two bits to represent one bit of actual information, we avoid the problem that single bits have the same value on every path. In this case, off-the-shelf static analyzers can no longer track the precise value of any variable.

Of course, given the knowledge of our scheme, the defender has always the option to adapt the analysis such that the used encoding is taken into account. Similar to before, it would be possible to keep the exact values for those variables that encode the same value after each loop iteration. However, this would require special treatment of the particular encoding scheme in use. Our experimental results demonstrate that the simple opaque constant calculation is already sufficient to thwart current malware detectors. However, we also explored the design space of opaque constants to identify primitives for which stronger guarantees with regard to robustness against static analysis can be provided. In the following paragraphs, we discuss a primitive that relies on the NP-hardness of the 3-satisfiability problem.

NP-Hard Opaque Constant Calculation The idea of the following opaque constant is that we encode the instance of an NP-hard problem into a code sequence that calculates our desired constant. That is, we create an opaque constant such that the generation of an algorithm to precisely determine the result of the code sequence would be equivalent to finding an algorithm to solve an NP-hard problem. For our primitive, we have chosen the 3-satisfiability problem (typically abbreviated as *3SAT*) as a problem that is known to be hard to solve. The 3SAT problem is a decision problem where a formula in Boolean logic is given in the following form:

$$\bigwedge_{i=1}^{n}(V_{i1} \vee V_{i2} \vee V_{i3})$$

where $V_{ij} \in \{v_1, ..., v_m\}$ and $v_1, ..., v_m$ are Boolean variables whose value can be either *true* or *false*. The task is now to determine if there exists an assignment for the variables v_k such that the given formula is satisfied (i.e., the formula evaluates to true). 3SAT has been proven to be NP-complete in [9].

Consider the code sequence in Figure 2. In this primitive, we define m boolean variables $v_1 \dots v_m$, which correspond directly to the variables in the given 3SAT formula. By $\overline{v_1} \dots \overline{v_m}$, we denote their negations. The pointers V_{11} to V_{n3} refer to the variables used in the various clauses of the formula. In other words, the pointers V_{11} to V_{n3} encode a 3SAT problem based on the variables $v_1 \dots v_m$. The loop simply evaluates the encoded 3SAT formula on the input. If the assignment of variables $v_1 \dots v_m$ does not satisfy the formula, there will always be at least one clause i that evaluates to false. When the check in the loop is evaluated for that specific clause, the result will always be true (as the check is performed against the negate of the clause). Therefore, the opaque constant will be set to 0. On the other hand, if the assignment satisfies the encoded formula, the check performed in the loop will never be true. Therefore, the value of the opaque constant is not overwritten and remains 1.

In the opaque constant presented in Figure 2, the 3SAT problem (that is, the pointers V_{11} to V_{n3}) is prepared by the obfuscator. However, the actual assignment of boolean values to the variables $v_1 \dots v_m$ is randomly performed during runtime. Therefore, the analyzer cannot immediately evaluate the formula. The trick of our opaque constant is that the

3SAT problem is prepared such that the formula is not satisfiable. Thus, independent of the actual input, the constant will always evaluate to 0. Of course, when a constant value of 1 should be generated, we can simply invert the result of the satisfiability test. Note that it is possible to efficiently generate 3SAT instances that are not satisfiable with a high probability [16]. A static analyzer that aims to exactly determine the possible values of our opaque constant has to solve the instance of the 3SAT problem. Thus, 3SAT is reducible in polynomial time to the problem of exact static analysis of the value of the given opaque constant.

Note that the method presented above only generates one bit of opaque information but can be easily extended to create arbitrarily long constants.

Basic Block Chaining One practical drawback of the 3SAT primitive presented above is that its output has to be the same for all executions, regardless of the actual input. As a result, one can conceive an analysis technique that evaluates the opaque constant function for a few concrete inputs. When all output values are equal, one can assume that this output is the opaque value encoded. To counter this analysis, we introduce a method that we denote *basic block chaining*.

With basic block chaining, the input for the 3SAT problems is not always selected randomly during runtime. Moreover, we do not always generate unsatisfiable 3SAT instances, but occasionally insert also satisfiable instances. In addition, we ensure that the input that solves a satisfiable formula is provided during runtime. To this end, the input variables $v_1 \dots v_m$ to the various 3SAT formulas are realized as global variables. At the end of every basic block, these global variables are set in one of the three following ways: (1) to static random values, (2) to random values generated at runtime, or (3), to values specially crafted such that they satisfy a solvable formula used to calculate the opaque constant in the *next* basic block in the control flow graph.

To analyze a program that is obfuscated with basic block chaining, the analyzer cannot rely on the fact that the encoded formula is always unsatisfiable. Also, when randomly executing a few sample inputs, it is unlikely that the analyzer chooses values that solve a satisfiable formula. The only way to dissect an opaque constant would be to first identify the basic block(s) that precede a certain formula and then determine whether the input values stored in this block satisfy the 3SAT problem. However, finding these blocks is not trivial, as the control flow of the program is obfuscated to make this task difficult (see the following Section 2.2 for more details). Thus, the analysis would have to start at the program entry point and either execute the program dynamically or resort to an approach similar to whole program simulation in which different branches are followed from the start, resolving opaque constants as the analysis progresses. Obviously, our obfuscation techniques fail against such methods, and indeed, this is consistent with an important point that we intend to make in this paper: dynamic analysis techniques are a promising and powerful approach to deal with obfuscated binaries.

2.2 Obfuscating Transformations

Using opaque constants, we possess a mechanism to load a constant value into a register without the static analyzer knowing its value. This mechanism can be expanded to perform a number of transformations that obfuscate the control flow, data locations, and data usage of a program.

2.2.1 Control Flow Obfuscation

A central prerequisite for the ability to carry out advanced program analysis is the availability of a *control flow graph*. A Control Flow Graph (CFG) is defined as a directed graph $G = (V, E)$ in which the vertices $u, v \in V$ represent basic blocks and an edge $e \in E : u \rightarrow v$ represents a possible flow of control from u to v. A basic block describes a sequence of instructions without any jumps or jump targets in the middle. More formally, a basic block is defined as a sequence of instructions where the instruction in each position dominates, or always executes before, all those in later positions. Furthermore, no other instruction executes between two instructions in the same sequence. Directed edges between blocks represent jumps in the control flow, which are caused by control transfer instructions (CTI) such as calls, conditional jumps, and unconditional jumps.

The idea to obfuscate the control flow is to replace unconditional jump and call instructions with a sequence of instructions that do not alter the control flow, but make it difficult to determine the target of control transfer instructions. In other words, we attempt to make it as difficult as possible for an analysis tool to identify the edges in the control flow graph. Jump and call instructions exist as direct and indirect variants. In case of a direct control transfer instruction, the target address is provided as a constant operand. To obfuscate such an instruction, it is replaced with a code sequence that does not immediately reveal the value of the jump target to an analyst. To this end, the substituted code first calculates the desired target address using an opaque constant. Then, this value is saved on the stack (along with a return address, in case the substituted instruction was a call). Finally, a `x86 ret(urn)` operation is performed, which transfers control to the address stored on top of the stack (i.e., the address that is pointed to by the stack pointer). Because the target address was previously pushed there, this instruction is equivalent to the original jump or call operation.

Typically, this measure is enough to effectively avoid the reconstruction of the CFG. In addition, we can also use obfuscation for the return address. When we apply this more complex variant to calls, they become practically indistinguishable from jumps, which makes the analysis of the resulting binary even harder because calls are often treated differently during analysis.

2.2.2 Data Location Obfuscation

The location of a data element is often specified by providing a constant, absolute address or a constant offset relative to a particular register. In both cases, the task of a static analyzer can be complicated if the actual data element that is accessed is hidden.

When accessing a global data element, the compiler typically generates an operation that uses the constant address of this element. To obfuscate this access, we first generate code that uses an opaque constant to store the element's address in a register. In a second step, the original operation is replaced by an equivalent one that uses the address in the register instead of directly addressing the data element. Accesses to local variables can be obfuscated in a similar fashion. Local variable access is typically achieved by using a constant offset that is added to the value of the base pointer register, or by subtracting a constant offset from the stack pointer. In both cases, this offset can be loaded into a register by means of an opaque constant primitive. Then, the now unknown value (from the point of view of the static analyzer) is used as offset to the base or stack pointer.

Another opportunity to apply data location obfuscation are indirect function calls and indirect jumps. Modern operating systems make heavy use of the concept of dynamically linked libraries. With dynamically linked libraries, a program specifies a set of library functions that are required during execution. At program start-up, the dynamic linker maps these requested functions into the address space of the running process. The linker then populates a table (called import table or procedure linkage table) with the addresses of the loaded functions. The only thing a program has to do to access a library function during runtime is to jump to the corresponding address stored in the import table. This "jump" is typically realized as an indirect function call in which the actual target address of the library routine is taken from a statically known address, which corresponds to the appropriate table entry for this function.

Because the address of the import table entry is encoded as a constant in the program code, dynamic library calls yield information on what library functions a program actively uses. Furthermore, such calls also reveal the important information of *where* these functions are called from. Therefore, we decided to obfuscate import table entry addresses as well. To this end, the import table entry address is first loaded into a register using an opaque constant. After this step, a register-indirect call is performed.

2.2.3 Data Usage Obfuscation

With data location obfuscation, we can obfuscate memory access to local and global variables. However, once values are loaded into processor registers, they can be precisely tracked. For example, when a function returns a value, this return value is typically passed through a register. When the value remains in the register and is later used as an argument to another function call, the static analyzer can establish this relationship. The problem from the point of view of the obfuscator is that a static analysis tool can identify *define-use-chains* for values in registers. That is, the analyzer can identify when a value is loaded into a register and when it is used later.

To make the identification of define-use chains more difficult, we obfuscate the presence of values in registers. To this end, we insert code that temporarily spills register content to an obfuscated memory location and later reloads it. This task is accomplished by first calculating the address of a temporary storage location in memory using an opaque constant. We then save the register to that memory location and delete its content. Some time later, before the content of the register is needed again, we use another opaque constant primitive to construct the same address and reload the register. For this process, unused sections of the stack are chosen as temporary storage locations for spilled register values.

After this obfuscation mechanism is applied, a static analysis can only identify two unrelated memory accesses. Thus, this approach effectively introduces the uncertainty of memory access to values held in registers.

3 Binary Transformation

To verify the effectiveness and robustness of the presented code obfuscation methods on real-world binaries, it was necessary to implement a binary rewriting tool that is capable of changing the code of arbitrary binaries without assuming access to source code or program information (such as relocation or debug information).

We did consider implementing our obfuscation techniques as part of the compiler tool-chain. This task would have been easier than rewriting existing binaries, as the compiler has full knowledge about the code and data components of a program and could insert obfuscation primitives during code generation. Unfortunately, using a compiler-based approach would have meant that it would not have been possible to apply our code transformations to real-world malware (except the few for which source code is available on the net). Also, the ability to carry out transformations directly on binary programs highlights the threat that code obfuscation techniques pose to static analyzers. When a modified compiler is required for obfuscation, a typical argument that is brought forward is that the threat

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.