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

1063-9527/07 $25.00 © 2007 IEEE
DOI 10.1109/ACSAC.2007.21

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 $v_1 \ldots v_m$ 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 is hypothetical because it is difficult to bundle a complete compiler with a malware program. In contrast, shipping a small binary rewriting engine together with malicious code is more feasible for miscreants.

When we apply the transformations presented in this paper to a binary program, the structure of the program changes significantly. This is because the code that is being rewritten requires a larger number of instructions after obfuscation, as single instructions get substituted by obfuscation primitives. To make room for the new instructions, the existing code section is expanded and instructions are shifted. This has important consequences. First, instructions that are targets of jump or call operations are relocated. As a result, the operands of the corresponding jump and call instructions need to be updated to point to these new addresses. Note that this also effects relative jumps, which do not specify a complete target address, but only an offset relative to the current address. Second, when expanding the code section, the adjacent data section has to be moved too. Unfortunately for the obfuscator, the data section often contains complex data structures that define pointers that refer to other locations inside the data section. All these pointers need to be adjusted as well.

Before instructions and their operands can be updated, they need to be identified. At first glance, this might sound straightforward. However, this is not the case because the variable length of the `x86` instruction set and the fact that code and data elements are mixed in the code section make perfect disassembly a difficult challenge.

In our system, we use a recursive traversal disassembler. That is, we start by disassembling the program at the program entry point specified in the program header. We disassemble the code recursively until every reachable procedure has been processed. After that, we focus on the remaining unknown sections. For these, we use a number of heuristics to recognize them as possible code. These heuristics include the use of byte signatures to identify function prologues or jump tables. Whenever a code region is identified, the recursive disassembler is restarted there. Otherwise, the section is declared as data.

Our rewriting tool targets both the Linux ELF and the Windows PE file formats. Using the recursive disassembler approach and our heuristics, our binary rewriting tool is able to correctly obfuscate many (although not all) real-world binaries. More detailed results on the robustness of the tool are provided in Section 4.

4 Evaluation

In this section, we present experimental results and discuss our experiences with our obfuscation tool. In particular, we assess how effective the proposed obfuscation techniques are in evading malware detectors. In addition, we analyze the robustness of our binary rewriting tool by processing a large number of Linux and Windows applications.

4.1 Evasion Capabilities

To demonstrate that the presented obfuscation methods can be used to effectively change the structure of a binary so that static analysis tools fail to recognize the obfuscated code, we conducted tests with real-world malware. We used our tool to morph three worm programs and then analyzed the obfuscated binaries using an advanced static analysis tool [10] as well as four popular commercial virus scanners.

The malware samples that we selected for our experiments were the `A` and `F` variants of the `MyDoom` worm and the `A` variant of the `Klez` worm. We chose these samples because they were used in the evaluation of the advanced static analysis tool in [10]. Thus, the tool was equipped with appropriate malware specifications to detect these worms. In order to obfuscate the malicious executables, we deployed the evasion techniques introduced in Section 2 using both the simple opaque constants and the one based on the 3SAT problem.

Commercial Virus Scanners: First, we tested the possibilities to evade detection by popular virus scanners. To evaluate the effectiveness of our obfuscation methods, we selected the following four popular anti-virus applications: McAfee Anti-Virus, Kaspersky Anti-Virus Personal, AntiVir Personal Edition, and Ikarus Virus Utilities.

Before the experiment, we verified that all scanners correctly identified the worms. Then, we obfuscated the three malicious code samples, ensured that the malware was still operating correctly, and ran the virus scanners on them. The results are shown in Table 1. In this table, an "X" indicates that the scanner was no longer able to detect the malware.

	`Klez.A`	`MyDoom.A`	`MyDoom.AF`
McAfee		X	
Kaspersky	X	X	X
AntiVir			X
Ikarus	X	X	X

Table 1. Evasion results for four commercial virus scanners

The results demonstrate that after the obfuscation process, the scanners from Kaspersky and Ikarus were not able to detect any of the malware instances. Surprisingly for us, however, the scanners from McAfee and AntiVir were still able to detect two out of three worms. Closer examination revealed that the scanner from McAfee detects the two obfuscated samples because of a virus signature that is based on parts of the data section. When we overwrote the bytes in the data section that were being used as a signature, the McAfee scanner could neither detect the original nor the obfuscated version of the malware anymore. In contrast, the AntiVir scanner uses a *combination* of both a data and a code signature to detect the worms. We were able to track down the data signature for both `Klez.A` and `MyDoom.A` to a few bytes in the data section. If any of these bytes in the data section was modified in the obfuscated binary, the detection by the virus scanner was successfully evaded. Indeed, it is relatively easy for malicious code to encrypt the data section using a different key for each instance. Hence, data signatures are not too difficult to evade.

Advanced Malware Detection (Model Checking): Because it is widely known that existing commercial virus scanners typically employ pattern-based signatures, the ability to evade their detection is not too surprising. In order to verify the efficiency of our obfuscation techniques on a more advanced malware detector, we obtained the system presented in [10] from its authors. This detector first creates a disassembly of the binary under analysis by using IDA Pro [7]. Then, model checking is used to search for the existence of a generic code template that characterizes malicious behavior. In particular, the tool attempts to identify code sequences in the program that copy this program's binary to another location in the file system. More precisely, a malicious code sequence is defined as a call to the `GetModuleFileNameA` Windows API function, followed by an invocation of the `CopyFileA` function. The exact specification as presented in [10] is shown below.

```
EF(%syscall(GetModuleFileNameA, $*,
          $pFile, 0) &
E %noassign($pFile) U
  %syscall(CopyFileA, $pFile))
```

Note that this specification requires that the same variable (`pFile`) is used as parameter in both function calls, without being overwritten in between. Because the malware detector uses a signature that characterizes the semantics of a sequence of code, it is resilient to code modifications that change the layout (e.g., register renaming or code insertion).

We first verified that the malicious code detector was able to correctly identify the three original worms and then applied our code transformations. After obfuscation, the tool was no longer able to identify any of the three malware instances. We examined in detail how our code transformations contributed to the successful evasion.

The first problem for the malware detector is its dependency on IDA Pro. After we obfuscated direct call and jump instructions, the recursive disassembler was no longer able to follow the control flow of the application. In this situation, IDA Pro reverts to a linear sweep analysis, which results in many disassembly errors. In fact, the output has such a poor accuracy that the library calls cannot be identified anymore. When we disable our control flow obfuscation transformations, IDA Pro produces a correct disassembly. However, the used detection signature relies on the fact that the dynamically linked Windows API functions `GetModuleFileNameA` and `CopyFileA` can be correctly identified. When we employ data location obfuscation, the analyzer can no longer determine which entry of the import table is used for library calls. Thus, the second problem is that the detection tool can no longer resolve the library function calls that are invoked by the malicious code. Assuming that library calls could be recognized, the malware detector would *still* fail to identify the malicious code. This is because the signature needs to ensure that the same parameter `pFile` is used in both calls. In our worm samples, this parameter was stored as a local variable on the stack. Again, using data location obfuscation, we can hide the value of the offset that is used together with the base pointer register to access this local variable. As a result, the static analysis tool cannot verify that the same parameter is actually used for both library calls, and detection fails.

Semantics-Aware Malware Detection: Another system that uses code templates instead of patterns to specify malicious code was presented in [3]. The first problem clearly is the dependency on IDA Pro, which produces incorrect disassembly output when confronted with control flow obfuscation. A second problem is the dependency of some code templates (or semantic signatures) on the fact that certain constants must be recognized as equivalent. Consider the template that specifies a decryption loop, which describes the behavior of programs that unpack or decrypt themselves to memory. According to [3], such a template consists of "(1) a loop that processes data from a source memory area and writes data to a destination memory area, and (2) a jump that targets the destination area." Clearly, the detector must be able to establish a relationship between the memory area where the code is written to and the target of the jump. However, when using data location obfuscation, the detector cannot statically determine where data is written to, and by using obfuscated jumps, it also cannot link this memory area with the target of the control flow instruction. Finally, semantic signatures can make use of define-use chains to link the location where a variable is set and the location where it is used. By using data usage obfuscation, however, such define-use chains can be broken.

4.2 Transformation Robustness

In this section, we discuss the robustness of the applied modifications as well as their size and performance impact. When testing whether obfuscation was successful, one faces the problem of test coverage. That is, it is not trivial to demonstrate that the obfuscated program behaves exactly like the original one. Because we operate directly on binaries, our biggest challenge is the correct distinction between code and data regions. When the disassembly step confuses code and data, addresses are updated incorrectly and the program crashes. We observed that disassembler errors quickly propagate through the program. Thus, whenever the binary rewriting fails, the obfuscated programs typically crash quickly. On the other hand, once an obfuscated application was running, we observed few problems during the more extensive tests we conducted. Thus, the mere fact that a program can be launched provides a good indication for the success of the transformation process. Of course, this is no guarantee for the correctness of the obfuscation process in general.

Linux Binaries In general, rewriting ELF binaries for Linux works very well. Our first experiment was performed on the GNU coreutils. This software package consists of 93 applications that can be found on virtually every Linux machine. Part of the coreutils package is a test script that performs 210 checks on various applications. To assess the robustness of our transformations, we rewrote all 93 applications using all obfuscation transformations introduced previously. We then ran the test script, and all 210 checks were passed without problems.

As a second experiment, we obfuscated all applications in the `/usr/bin/` directory on a machine running Ubuntu Linux 5.10. For this test, we rewrote 774 applications. When manually checking these applications, we recorded eleven programs that crashed with a segmentation fault. Among these programs were large, complex applications such as Emacs and Evolution or the linker. Of those programs that were successfully rewritten, we extensively used and tested applications such as the instant messenger gaim (806 KB), vim (1,074 KB), xmms (991 KB) and the Opera web browser (12,059 KB).

Windows Binaries The set of programs that we used for testing Windows executables consisted of twelve executables selected from the `%System%` directory, and the Internet Explorer. The selected applications were both GUI and command-line programs and represent a comprehensive set of applications, ranging from system utilities (ping) to editors (NotePad) and games (MS Hearts). After obfuscation and manual testing of their functionality, we could not identify any problems for eleven of the thirteen applications.

One of those two applications that worked only partially was the Windows Calculator. When our binary rewriting tool processes the calculator, an exception handler is not patched correctly. This causes a jump to an incorrect address whenever an exception is raised. That is, the obfuscated program calculates correctly. However, when a division by zero is executed, the application crashes. The second application that could not be obfuscated properly was the Clipboard. This application starts and can be used to copy text between windows. Unfortunately, when a file is copied to the Clipboard, the application appears to hang in an infinite loop.

4.3 Size and Performance

Typically, the most important goal when obfuscating a binary is to have it resist analysis, while size and performance considerations are only secondary. Nevertheless, to be usable in practice, the increase in size or loss in performance cannot be completely neglected.

We measured the increase of the code size when obfuscating the Linux binaries under `/usr/bin`. As the obfuscation transformations are applied to Windows and Linux executables in a similar fashion, the results for PE files are comparable. For the Linux files, the average increase of the code size was 237%, while the maximum increase was 471%, when we only used the simple loops for hiding constant values. When we used code that evaluates 3SAT formulas, the size of the binaries increased significantly more. For example, when using large 3SAT instances with more than 200 clauses, the code size sometimes increased by a factor as large as 30. Of course, when performing obfuscation, one can make a number of trade-offs to reduce the code size, for example, by sparse usage of the most space consuming transformations. However, even when applying the full range of obfuscation methods, a malware author will hardly be deterred by a huge size increase of his program.

During obfuscation, single instructions are frequently replaced by long code sequences. Nevertheless, the overall runtime of the obfuscated binaries did not increase dramatically, and we observed no noticeable difference for applications such as Opera or Internet Explorer. We then performed a series of micro-benchmarks with CPU-intense programs (such as grep, md5sum and zip) and found an average increase in runtime of about 50%. In the worst case, we observed a runtime that almost doubled, which is acceptable in many cases (especially for malware that is running on someone else's computer). With regards to performance, code that evaluates unsatisfiable 3SAT formulas is not slower than the simple opaque constants. The reason is that for nearly all random inputs, only very few clauses have to be considered before it is clear that the given input does not satisfy the 3SAT instance. On average, we observed that less than 7 clauses were evaluated before the constant can be determined. Again, we want to stress that performance is not a huge issue for most malicious programs.

4.4 Possible Countermeasures

In this paper, we describe techniques that make binaries more resistant to static analysis. Such techniques have not been encountered in the wild yet. However, it is well-known that malware authors are constantly working on the creation of more effective obfuscation and evasion schemes. Thus, we believe that it is important to explore future threats to be able to develop defenses proactively.

One possibility to counter our presented scheme is to flag programs as suspicious when they exhibit apparent signs of obfuscation. For example, when our control flow transformations are applied, the resulting code will contain many return instructions, but no call statements. Hence, even though the code cannot be analyzed precisely, it could be recognized as malicious. Unfortunately, when flagging obfuscated binaries as malicious, false positives are possible. The reason is that obfuscation may also be used for legitimate purposes, for example, to protect intellectual property.

A more promising approach when analyzing obfuscated binaries is to use dynamic techniques. As a matter of fact, most obfuscation transformations become ineffective once the code is executed. Hence, we believe that future malware analysis approaches should be centered around dynamic techniques that can effectively analyze the code that is run.

5 Related Work

The two areas that are most closely related to our work are code obfuscation and binary rewriting. Code obfuscation describes techniques to make it difficult for an attacker to extract high-level semantic information from a program [6, 20]. This is typically used to protect intellectual property from being stolen by competitors or to robustly embed watermarks into copyrighted software [5]. Similar to our work, researchers proposed obfuscation transformations that are difficult to analyze statically. One main difference to our work is that these transformations are applied to source code. Source code contains rich program information that make it easier to apply obfuscating operations.

In [6], opaque predicates were introduced, which are boolean expressions whose truth value is known during obfuscation time but difficult to determine statically. The idea of opaque predicates has been extended in this paper to hide constants, the basic primitive on which our obfuscation transformations rely. The one-way translation process introduced in [19, 20] is related to our work as it attempts to obscure control flow information by converting direct jumps and calls into corresponding indirect variants. The difference is the way control flow obfuscation is realized and the fact that we also target data location and data usage information. An obfuscation approach that is orthogonal to the techniques outlined above is presented in [13]. Here, the authors exploit the fact that it is difficult to distinguish between code and data in `x86` binaries and attempt to attack directly the disassembly process.

We are aware of two other pieces of work that deal with program obfuscation on the binary level. In [2], the authors developed a simple, binary obfuscator to test their malware detector. This obfuscator can apply transformations such as code reordering, register renaming, and code insertion. However, based on their description, a more powerful static analyzer such as the one introduced by the same authors in [3] can undo these obfuscations. In [21], a system is proposed that supports opaque predicates in addition to code reordering and code substitution. However, the control flow information is not obscured, and data usage and location information can be extracted. Thus, even if the opaque predicate cannot be resolved statically, a malware detector can still analyze and detect the branch that contains the operations of the malicious code.

In [1], the authors discussed the theoretical limits of program obfuscation. In particular, they prove that it is impossible to hide certain properties of particular families of functions using program obfuscation. In our work, however, we do not try to completely conceal all properties of the obfuscated code. Instead, we obfuscate the control flow between functions and the location of data elements and make it hard for static analysis to undo the process.

Besides program obfuscation, binary rewriting is the second area that is mostly related to this research. Static binary rewriting tools are systems that modify executable programs, typically with the goal of performing (post-link-time) code optimization or code instrumentation. Because these tools need to be safe (i.e., they must not perform modifications that break the code), they require relocation information to distinguish between address and non-address constants. To obtain the required relocation information, some tools only work on statically linked binaries [15], demand modifications to the compiler tool-chain [14], or require a program database (PDB) [17, 18]. Unfortunately, relocation information is not available for malicious code in the wild, thus, our approach sacrifices safety to be able to handle binaries for which no information is present.

Besides those tools that require relocation information, a few systems have been proposed that can process binary programs without relying on additional program information [12, 4]. These systems operate on RISC binaries, which is a significantly simpler task than working on the complex `x86` instruction set. Finally, binary rewriting has already been introduced by malicious code as a means to evade detection by virus scanners. The infamous Mistfall engine [22] is capable of relocating instructions of a program that is to be infected. Interestingly, the author of the Mistfall engine states that his rewriting algorithm fails to correctly patch the code for jump tables that are very common in windows binaries. In our implementation, we use a heuristic that allows us to correctly rewrite many binaries for which the Mistfall algorithm produces incorrect code.

6 Conclusions

In this paper, our aim was to explore the odds for a malware detector that employs powerful static analysis to detect malicious code. To this end, we developed binary program obfuscation techniques that make the resulting binary difficult to analyze. In particular, we introduced the concept of opaque constants, which are primitives that allow us to load a constant into a register so that the analysis tool cannot determine its value. Based on opaque constants, we presented a number of obfuscation transformations that obscure program control flow, disguise access to variables, and block tracking of values held in processor registers.

To be able to assess the effectiveness of such an obfuscation approach, we developed a binary rewriting tool that allows us to perform the necessary modifications. Using the tool, we obfuscated three well-known worms and demonstrated that neither virus scanners nor a more advanced static analysis tool based on model checking could identify the transformed programs.

While it is conceivable to improve static analysis to handle more advanced obfuscation techniques, there is a fundamental limit in what can be decided statically. In particular, we presented a construct based on the 3SAT problem that is provably hard to analyze. Limits of static analysis are of less concern when attempting to find bugs in benign programs, but they are more problematic and worrisome when analyzing malicious, binary code that is deliberately designed to resist analysis. In this paper, we demonstrate that static techniques alone might not be sufficient to identify malware. Indeed, we believe that such approaches should be complemented by dynamic analysis, which is significantly less vulnerable to code obfuscating transformations.

References

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In *Advances in Cryptology (CRYPTO)*, 2001.

[2] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Usenix Security Symposium*, 2003.

[3] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware Malware Detection. In *IEEE Symposium on Security and Privacy*, 2005.

[4] C. Cifuentes and M. V. Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *IEEE Computer*, 33(3), 2000.

[5] C. Collberg and C. Thomborson. Software Watermarking: Models and Dynamic Embeddings. In *ACM Symposium on Principles of Programming Languages*, 1999.

[6] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Conference on Principles of Programming Languages (POPL)*, 1998.

[7] Data Rescure. IDA Pro: Disassembler and Debugger. `http://www.datarescue.com/idabase/`, 2006.

[8] L. Gordon, M. Loeb, W. Lucyshyn, and R. Richardson. Computer Crime and Security Survey. Technical report, Computer Security Institute (CSI), 2005.

[9] R. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, 1972.

[10] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting Malicious Code by Model Checking. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2005.

[11] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Annual Computer Security Application Conference (ACSAC)*, 2004.

[12] J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[13] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[14] L. V. Put, D. Chanet, B. D. Bus, B. D. Sutter, and K. D. Bosschere. Diablo: A reliable, retargetable and extensible link-time rewriting framework. In *IEEE International Symposium On Signal Processing And Information Technology*, 2005.

[15] B. Schwarz, S. Debray, and G. Andrews. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *Workshop on Binary Translation (WBT)*, 2001.

[16] B. Selman, D. Mitchell, and H. Levesque. Generating hard satisability problems. *Artificial Intelligence*, 81(1 – 2), 1996.

[17] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[18] A. Srivastava and H. Vo. Vulcan: Binary transformation in distributed environment. Technical report, Micorosft Research, 2001.

[19] C. Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia, 2001.

[20] C. Wang, J. Hill, J. Knight, and J. Davidson. Protection of Software-Based Survivability Mechanisms. In *International Conference on Dependable Systems and Networks (DSN)*, 2001.

[21] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, 2002.

[22] Z0mbie. Automated reverse engineering: Mistfall engine. VX heavens, `http://vx.netlux.org/lib/vzo21.html`, 2006.