Impeding Malware Analysis Using Conditional Code Obfuscation

Monirul Sharif[1] Andrea Lanzi[2] Jonathon Giffin[1] Wenke Lee[1]

[1]School of Computer Science, College of Computing, Georgia Institute of Technology, USA
{msharif, giffin, wenke}@cc.gatech.edu

[2]Dipartimento di Informatica e Comunicazione, Università degli Studi di Milano, Italy
andrew@security.dico.unimi.it

Abstract

Malware programs that incorporate trigger-based behavior initiate malicious activities based on conditions satisfied only by specific inputs. State-of-the-art malware analyzers discover code guarded by triggers via multiple path exploration, symbolic execution, or forced conditional execution, all without knowing the trigger inputs. We present a malware obfuscation technique that automatically conceals specific trigger-based behavior from these malware analyzers. Our technique automatically transforms a program by encrypting code that is conditionally dependent on an input value with a key derived from the input and then removing the key from the program. We have implemented a compiler-level tool that takes a malware source program and automatically generates an obfuscated binary. Experiments on various existing malware samples show that our tool can hide a significant portion of trigger based code. We provide insight into the strengths, weaknesses, and possible ways to strengthen current analysis approaches in order to defeat this malware obfuscation technique.

1 Introduction

With hundreds of new malware samples appearing every day [3], malware analysis, which attempts to understand and extract the capabilities or behavior from malware samples, is becoming increasingly important. As malware analysis techniques evolve, malware writers continually employ sophisticated anti-reverse engineering techniques in an effort to defeat and evade the state-of-the-art analyzers. Historically, encryption [44], polymorphism [31, 39], and other obfuscation schemes [11,12,25,41,43] have been primarily employed to thwart anti-virus tools and static analysis [9,10,22,23] based approaches. Dynamic analysis based approaches [2, 6, 13, 29, 40] inherently overcome all anti-static analysis obfuscations, but they only observe a single execution path. Malware can exploit this limitation by employing trigger-based behaviors such as time-bombs, logic-bombs, bot-command inputs, and testing the presence of analyzers, to hide its intended behavior.

Recent analyzers provide a powerful way to discover trigger based malicious behavior in arbitrary malicious programs. Moser et al. proposed a scheme [28] that explores multiple paths during execution of a malware. After exploring a branch, their technique resumes execution from a previously saved state and takes the alternate branch by inverting the condition and solving constraints to modify related memory variables in a consistent manner. Other recently proposed approaches can make informed path selection [4], discover inputs that take a specific path [5, 7, 13] or force execution along different paths [42]. We call all such approaches *input-oblivious analyzers* because they do not utilize any source of information about inputs other than the program being analyzed.

Our goal is to anticipate attacks against the state-of-the-art malware analyzers in order to develop more effective analysis techniques. We present a simple, automated and transparent obfuscation against powerful input oblivious analyzers. We show that *it is possible to automatically conceal trigger-based malicious behavior of existing malware from any static or dynamic input-oblivious analyzer by an automatically applicable obfuscation scheme based on static analysis*.

Our scheme, which we call *conditional code obfuscation*, relies on the principles of secure triggers [18]. First, we identify and transform specific branch conditions that rely on inputs by incorporating one-way hash functions in such a way that it is hard to identify the values of variables for which the conditions are satisfied. Second, the *conditional code*, which is the code executed when these conditions are satisfied, is identified and encrypted with a key that is derived from the value that satisfies the condition. As a result, input oblivious analyzers can no longer feasi-

bly determine the values that satisfy the condition and consequently the key to unveil the conditional code. Our approach utilizes several static analysis techniques, including control dependence analysis, and incorporates both source-code and binary analysis to automate the entire process of transforming malware source programs to their obfuscated binary forms.

In order to show that conditional code obfuscation is a realistic threat, we have developed a compiler-level tool that applies the obfuscation to malware programs written in C/C++. Our prototype implementation generates obfuscated compiled ELF binaries for Linux. Since the malware authors will be the ones applying this technique, the assumption of having the source code available is realistic. We have tested our system by applying it on several real malware programs and then evaluated its effectiveness in concealing trigger based malicious code. In our experiments on 7 different malware programs containing 92 malicious triggers, our tool successfully obfuscated and concealed the entire code that implemented 87 of them.

We analyze the strengths and weaknesses of our obfuscation. Although the keys are effectively removed from the program, the encryption is still susceptible to brute force and dictionary attacks. We provide a method to measure the strength of particular applications of the obfuscation against such attacks. To understand the possible threats our proposed obfuscation scheme poses, we discuss how malware authors may manually modify their code in different ways to take advantage of the obfuscation technique. Finally, we provide insight into possible ways of defeating the obfuscation scheme, including more informed key search attacks and the incorporation of input-domain information in existing analyzers.

We summarize our contributions below:

- We present the principles of an automated obfuscation scheme that can conceal condition-dependent malicious behavior from existing and future input oblivious malware analyzers.

- We have developed a prototype compiler-level obfuscator for Linux and performed experimental evaluation on several existing real-world malware programs, showing that a large fraction of trigger based malicious behavior can be successfully hidden.

- We provide insight into the strengths and weaknesses of our obfuscation. We discuss how an attacker equipped with this knowledge can modify programs to increase the strength of the scheme. We also discuss the possibility of brute-force attacks as a weakness of our obfuscation and provide insight into how to develop more capable malware analyzers that incorporate input domain knowledge.

Unlike polymorphic code, our approach does not store encryption keys inside the program. However, this does not limit the usage of polymorphism on our obfuscated binaries. It can be added as a separate layer on top of our obfuscation.

The goal of our obfuscation is to hide malicious behavior from malware analyzers that extract behavior. For these analyzers, the usual assumption is that the program being analyzed is already suspicious. Nevertheless, malware authors wish to develop code that is not easily detected. Naïve usage of this obfuscation may actually improve malware detection because of the particular way in which hash functions and decryption routines are used. However, since attackers can add existing polymorphic or metamorphic obfuscation techniques on top of our technique, a detector should be able to detect such malware at best with the same efficacy as polymorphic malware detection.

2 Conditional Code Obfuscation

Malware programs routinely employ various kinds of trigger based events. The most common examples are bots [14], which wait for commands from the botmaster via a command and control mechanism. Some keyloggers [37] log keys from application windows containing certain keywords. Timebombs [27] are malicious code executed at a specific time. Various anti-debugging [8] or anti-analysis tricks detect side-effects in the executing environment caused by analyzers and divert program execution when present. The problem for the malware writer is that the checks inside the program that are performed on the inputs give away information about what values are expected. For example, the commands a bot supports are usually contained inside the program as strings. More generally, for any trigger based behavior, the conditions recognizing the trigger reveal information about the inputs required to activate the behavior.

The basis of our obfuscation scheme is intuitive. By replacing input-checking conditions with equivalent ones that recognize the inputs without revealing information about them, the inputs can become secrets that the input-oblivious analyzer can no longer discover. Such secrets can then be used as keys to encrypt code. Since the modified conditions are satisfied only when the inputs are sent to the program, the code blocks that are conditionally executed can be encrypted. In other words, our scheme encrypts conditional code with a key that is removed from the program, but is evident when the modified condition is satisfied. Automatically carrying out this transformation as a general obfuscation scheme involves several subtle challenges.

We provide a high-level overview of our obfuscation with program examples in Section 2.1. The general mechanism is defined in Section 2.2. The program analysis algorithms and transformations required are described in Sec-

```
cmd  = get_command(sock);
if (strcmp(cmd,"startkeylogger") == 0)
{
    log_keys();
}
```

```
n = get_day_of_month();
if ((n > 10) && (n < 20))
{
    attack();
}
```

Figure 1. Two conditional code snippets.

tion 2.3. Section 2.4 describes the consequences of our scheme on existing malware analysis approaches. Section 2.5 discusses possible brute-force attacks on our obfuscation technique.

2.1 Overview

Figure 1 shows snippets of two programs that have conditionally executed code. The first program snippet calls a function that starts logging keystrokes after receiving the command "startkeylogger". The second example starts an attack only if the day of month is between the 11th and the 19th. In both the programs, the expected input can be easily discovered by analyzing the code.

We use cryptographic hash functions to hide information. For the first example, we can modify the condition to compare the computed the hard-coded hash of the string in `cmd` with the hash value of the string "startkeylogger" (Figure 2). The command string "startkeylogger" becomes a secret that an input oblivious analyzer cannot know. This secret can be used as the key to encrypt the conditional code block and the entire function `log_keys()`. Notice that when the expected command is received, the execution enters the `if` block and the encrypted block is correctly decrypted and executed.

```
cmd  = get_command(sock);
if (hash(cmd) == H)
    /* here, H = hash("startkeylogger") */
{
    decrypt_function(encr_log_keys, cmd);
    encr_log_keys();  /* encrypted log_keys */
}
```

Figure 2. Obfuscated example snippet.

In the second example of Figure 1, cryptographic hashes of the operands do not provide a condition equivalent to the original. Moreover, since several values of the variable n satisfy the condition, it is problematic to use them as keys for encryption and decryption.

We define *candidate* conditions as those suitable for our obfuscation. A candidate needs three properties. First, the ordering relation between the pre-images of the hash function must be maintained in the images. Second, there should be a unique key derived from the condition when it is satisfied. Third, the condition must contain an operand that has a statically determinable constant value.

Given these requirements above, operators that check equality of two data values are suitable candidates. Hence, conditions having '==', `strcmp`, `strncmp`, `memcmp`, and similar operators can be obfuscated with our mechanism.

2.2 General Mechanism

We now formally define the general method of our conditional code obfuscation scheme. Without loss of generality, we assume that any candidate condition is equivalent to the simple condition "$X == c$" where the operand c has a statically determinable constant value and X is a variable. Also, suppose that a code block B is executed when this condition is satisfied. Figure 3 shows the program transformation required for the obfuscation. The cryptographic hash function is denoted by $Hash$ and the symmetric encryption and decryption routines are $Encr$ and $Decr$, respectively.

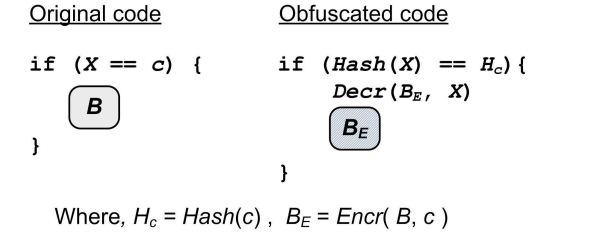


Figure 3. General obfuscation mechanism.

The obfuscated condition is "$Hash(X) == H_c$" where $H_c = Hash(c)$. The *pre-image resistance* property of the function $Hash$ implies that it is infeasible to find c given H_c. This ensures that it is hard to reverse the hash function. In addition, because of the *second pre-image resistance* property, it is hard to find another c' for which $Hash(c') = H_c$. Although this property does not strengthen the obfuscation, it is required to make the obfuscated condition semantically equivalent to the original, ensuring the correctness of the program.

The block B is encrypted with c as the key. Let B_E be the encrypted block where $B_E = Encr(B, c)$. Code is inserted immediately before B_E to decrypt it with the key contained in variable X. Since $Hash(X) = H_c$ implies $X = c$, when the obfuscated condition is satisfied, the

original code block is found, i.e. $B = Decr(B_E, c)$ and the program execution is equivalent to the original. However, a malware analyzer can recover the conditional code B only by watching for the attacker to trigger this behavior, by guessing the correct input, or by cracking the cryptographic operations.

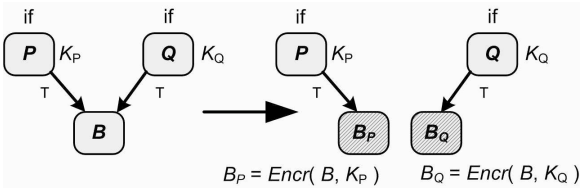


Figure 4. Duplicating conditional code.

2.3 Automation using Static Analysis

In order to apply the general mechanism presented in the previous section automatically on a program, we utilized several known algorithms in static analysis. In this section, we describe how these program analysis techniques were used.

2.3.1 Finding Conditional Code

In order to identify conditional code suitable for obfuscation, we first identify candidate conditions in a program. Let $\mathcal{F}$ be the set of all functions or procedures and $\mathcal{B}$ be the set of all basic blocks in the program that we analyze. For each function $F_i \in \mathcal{F}$ in the program, we construct a *control flow graph* (CFG) $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ in the program where $\mathcal{V}_i \subseteq \mathcal{B}$ is the set of basic blocks in F_i and $\mathcal{E}_i$ is the set of edges in the CFG representing control-flow between basic blocks. We then identify basic blocks having conditional branches, which have two outgoing edges. Since we are not interested in conditions used in loops, we employ loop analysis to identify such conditions and discard them. From the remaining conditional branches, we select candidate conditions as the ones containing equality operators as described in Section 2.1. Let $\mathcal{C}_i \subseteq \mathcal{V}_i$ be the set of blocks containing candidate conditions for each function F_i.

After candidate conditions are identified in a program, the next step is to find corresponding conditional code blocks. As described earlier, conditional code is the code that gets executed when a condition is satisfied. It may include some basic blocks from the same function the condition resides in and some other functions. Since a basic block can contain at most one conditional branch instruction, by a condition, we refer to the basic block that contains it. We use the mapping $CCode : \mathcal{B} \rightarrow (\mathcal{B} \cup \mathcal{F})^*$ to represent conditional code for any condition.

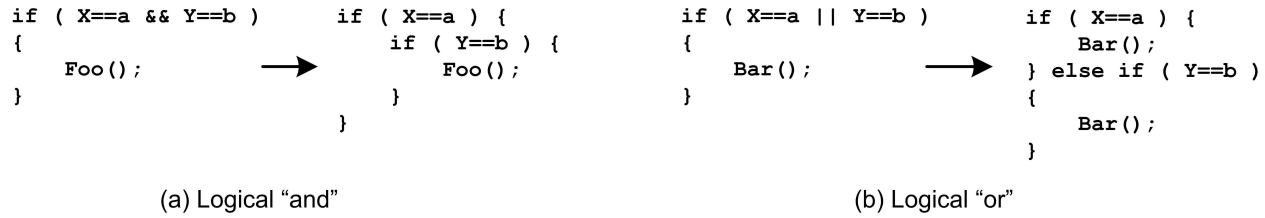
In order to determine conditional code, we first use *control dependence analysis* [1, 16] at the intra-procedural level. A block Q is control dependent on another block P if the outcome of P determines the reachability of Q. More precisely, if one outcome of P always executes Q, but the other outcome may not necessarily reach Q, then Q is control dependent on P. Using the standard algorithm for identifying control dependence, we build a *control dependence graph* (CDG) for each function, where each condition block and their outcome has edges to blocks that are control dependent on it. For each candidate condition, we find the set of blocks that are control dependent on its true outcome. Therefore, if a true outcome of a candidate condition $C \in \mathcal{C}_i$ of function F_i has an edge to a block $B \in \mathcal{V}_i$, then $B \in CCode(C)$.

Conditional code blocks may call other functions. To take them into account, we determine reachability in the inter-procedural CFG. If there exists a call to a function F from a conditional code block $B \in CCode(C)$ of some candidate condition C, we consider $F \in CCode(C)$ which means that we consider all the code in the function F as conditional code of C as well. Now, for every block in $F \in CCode(C)$, we find calls to other functions and include them in $CCode(C)$. This step is performed repeatedly until all reachable functions are included. We used this approach instead of inter-procedural control-dependence analysis [35] because it allows us to obfuscate functions that are not control-dependent on a candidate condition but can be reached only from that condition.

A candidate condition may be contained in a block that is conditional code of another candidate condition. The next step is to eliminate these cases by making a function or block conditional code of only the closest candidate condition that can reach it. For any block $B \in CCode(C)$, if $B \in CCode(C')$ where $C \neq C'$ and $C \in CCode(C')$ then we remove B from $CCode(C')$. We perform the same operation for functions.

Blocks and functions can be obfuscated when they are conditional code of candidate conditions only. If they are reachable by non-candidate conditions, then we cannot obfuscate them. When obfuscations are to be applied, they are applied in an iterative manner, starting with the candidate condition that have no other candidate conditions depending on it. The basic blocks and functions that are conditional code of these conditions are obfuscated first. In the next iteration, candidate conditions with no unobfuscated candidate conditions depending on it are obfuscated. The iterative process continues until all candidate conditions are obfuscated.

We use a conservative approach to identify conditional code when the CFG is incomplete. If code pointers are used whose targets are not statically resolvable, we do not encrypt code blocks that are potential targets and any other code blocks that are reachable from them. Otherwise, the



(a) Logical "and" (b) Logical "or"

Figure 5. Compound condition simplification.

encryption can crash the program. Fortunately, type information frequently allows us to limit the set of functions or blocks that are probable targets of a code pointer.

2.3.2 Handling Common Conditional Code

A single block or a function may be conditional code of more than one candidate condition that are not conditional code of each other. For example, a bot program may contain a common function that is called after receiving multiple different commands. If a block B is conditional code of two candidate conditions P and Q, where $P \notin CCode(Q)$ and $Q \notin CCode(P)$ then B can be reached via two different candidate conditions and cannot be encrypted with the same key. As shown in Figure 4, we solve this problem by duplicating the code and encrypting it separately for each candidate condition.

2.3.3 Simplifying Compound Constructs

Logical operators such as `&&` or `||` combine more than one simple condition. To apply our obfuscation, compound conditions must be first broken into semantically equivalent but simplified conditions. However, parts of a compound condition may or may not be candidates for obfuscation, making the compound condition unsuitable for obfuscation.

Logical `and` operators (`&&`) can be written as nested `if` statements containing the operand conditions and the conditional block in the innermost block (Figure 5(a)). Since both the simple conditions must be satisfied to execute conditional code, the code can be obfuscated if at least one of them is a candidate condition.

Logical `or` operators (`||`) can be obfuscated in two ways. Since either of the conditions may execute conditional code, the conditional code may be encrypted with a single key and placed in two blocks that are individually obfuscated with the simple conditions. Another simple way is to duplicate the conditional code and use `if...else if` constructs (Figure 5(b)). Note that if either one of the two conditions is not a candidate for obfuscation, then the conditional code will remain revealed. Concealing the other copy does not gain protection. Although it is not possible to determine that the concealed code is equivalent to the revealed one, the revealed code gives away the behavior that was intended to be hidden from an analyzer.

To introduce more candidate conditions in C/C++ programs, we convert `switch...case` constructs into several `if` blocks, each containing a condition using an equality operator. Every case except the *default* becomes a candidate for obfuscation. Complications arise when a code block under a switch case falls through to another. In such cases, code of the block in which control-flow falls through can be duplicated and contained in the earlier switch case code block, and then the standard approach that we described can be applied.

2.4 Consequences to Existing Analyzers

Our obfuscation can thwart different classes of techniques used by existing malware analyzers. The analysis refers to the notations presented in Section 2.2.

Path exploration and input discovery: Various analysis techniques have been proposed that can explore paths in a program to identify trigger based behavior. Moser et al.'s dynamic analysis based approach [28] explores multiple paths during execution by repeatedly restoring earlier saved program states and solving constructed path constraints in order to find a consistent set of values of in-memory variables that satisfy conditions leading to different paths. They use dynamic taint analysis on inputs from system calls to construct linear constraints representing dependencies among memory variables. After our obfuscation is applied, the constraint added to the system is "$Hash(X) == H_c$", which is a non-linear function. Therefore, our obfuscation makes it hard for such a multi-path exploring approach to feasibly find value assignments to variables in the programs's memory to proceed towards the obfuscated path.

A similar effect can be seen for approaches that discover inputs from a program that executes it along a specific path. EXE [7] uses mixed symbolic and concrete execution to create constraints that relate inputs to variables in memory. It uses its own efficient constraint solver called STP, which supports all arithmetic, logical, bitwise, and relational operators found in C (including non-linear operations). Cryptographic hash functions are designed to be computationally infeasible to reverse. Even with a powerful solver like

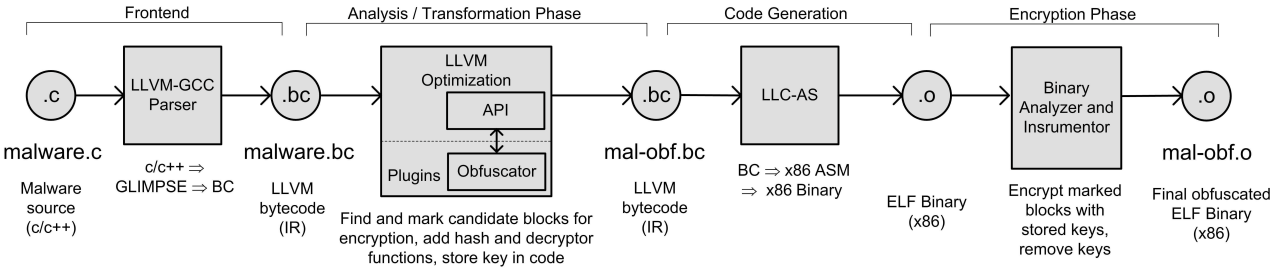


Figure 6. The architecture of our automated conditional code obfuscation system.

STP, it is infeasible to generate and solve the constraints that represent the complete set of operations required to reverse such a function.

Forcing execution: A malware analyzer may force execution along a specific path without finding a consistent set of values for all variables in memory [42], hoping to see some malicious behavior before the program crashes. Suppose that the analyzer forces the malware program to follow the obfuscated branch that would originally execute B without finding the key c. Assuming X has a value c' where $c' \neq c$, the decrypted block then becomes $B' = Decr(B_E, c') \neq B$. In practical situations, subsequent execution of B' should cause the program to eventually crash without revealing any behavior of the original block B.

Static analysis: The utilization of hash functions alone cannot impede approaches utilizing pure static analysis or hybrid methods [4] because behavior can be extracted by analyzing the code without requiring constraint solving. However, our utilization of encryption conceals the behavior in the encrypted blocks B_E that can only be decrypted by the key c, which is no longer present in the program.

2.5 Brute Force and Dictionary Attacks

Although existing input-oblivous techniques can be thwarted by our obfuscation, analyzers that are aware of the obfuscation may attempt brute-force attacks on the keys used for encryption. First, the hash function $Hash()$ being used in the malware needs to be extracted by analyzing the code. Then, a malware analyst may try to identify c by computing $Hash(X)$ for all suitable values of X and searching for a value satisfying $Hash(X) = H_c$. The strength of the obfuscation applied to the condition can therefore be measured by the size of the minimal set of suitable values of X.

Let $Domain(X)$ denote the set of all possible values that X may take during execution. If τ is the time taken to test a single value of X or the hash computation time, then the brute force attempt will take $|Domain(X)|\tau$ time. Finding the set $Domain(X)$ is not straightforward. In most cases, only the size of X may be known. If X is n bits in length, then the brute force attack requires $2^n\tau$ time. The possibility of using a pre-computed hash table to reduce search time can be thwarted by using a nonce with the data before computing the hash. Moreover, different nonce values for different conditions can be used to make the computed hash for one condition not useful for another.

Existing program analysis techniques such as data-flow analysis or symbolic execution may provide a smaller $Domain(X)$ in some cases, enabling a dictionary attack. Section 5 discusses more about attacks on our obfuscation, including automated techniques that can be incorporated in current analyzers.

3 Implementation Approach

In this section, we present the design choices and implementation of the compiler-level tool that we developed to demonstrate the automated conditional code obfuscation on malware programs. Our primary design challenge was to select the appropriate level of code on which to work on. Performing encryption at a level higher than binary code would cause un-executable code to be generated after decryption at run-time. On the other hand, essential high-level information such as data types require analysis at a higher level. Our system, therefore, works at both the intermediate code and the binary level.

We use the LLVM [24] compiler infrastructure and the DynInst [20] binary analysis and instrumentation system. The LLVM framework is an extensible program optimization platform providing an API to analyze and modify its own RISC-like intermediate code representation. It then emits binary code for various processor families. Most of the heavy-duty analysis and code instrumentation is done using the help of LLVM. The DynInst tool is a C/C++ based binary analysis and instrumentation framework, which we use for binary rewriting.

We implemented our prototype for the x86 architecture and the Linux OS. We targeted the Linux platform primarily to create a working system to showcase the capability of the proposed obfuscation scheme without making it a widely

applicable tool for malware authors. However, the architecture of our obfuscation tool is general enough to be portable to the Windows OS.

The architecture of our system is presented in Figure 6. Our system takes as input a malware source program written in C/C++ and generates an obfuscated binary in the Linux ELF format. The transformation is done in four phases. The phases are (1) the Frontend Code Parsing Phase, (2) the Analysis/Transformation Phase, (3) the Code Generation Phase, and (4) the Encryption Phase.

In the first phase LLVM's GCC-based parser converts C/C++ source programs to LLVM's intermediate representation. Next, in the analysis and transformation phase, the bulk of the obfuscation except for the actual encryption process is carried out on the intermediate code. In this phase, candidate conditions are identified and obfuscated, conditional code blocks are instrumented with decipher and marker routines, and the key required for encryption is stored as part of the code. Section 3.1 describes these steps in details. In the third phase, we use the static compilation and linking back end of LLVM to convert the LLVM intermediate representation (IR) to x86 assembly from which we generate an x86 ELF binary. In the final phase, which is described in Section 3.2, our DynInst based binary modification tool encrypts marked code blocks to complete the obfuscation process. We describe in Section 3.3 how the decryption of code takes place during run-time.

3.1 Analysis and Transformation Phase

The analysis and transformation was implemented as an LLVM plugin, which is loaded by the LLVM optimization module. The steps taken are illustrated in Figure 7 and described below.

3.1.1 Candidate Condition Replacement

We followed the method described in section 2.3 to identify candidate conditions and their conditional code. As shown in Figure 7, the transformed code calls the hash function with the variable used in the condition as the argument. We use SHA-256 in our implementation as the hash function. The replaced condition compares the result with the hard coded hash value of the constant. In other words, the condition "$X == c$" is replaced with "$Hash(X) == H_c$", where $H_c = Hash(c)$. Depending on the data type of X, calls are placed to different versions of the hash function. For length constrained string functions, the prefixes of the strings are taken. A special wrapper function is used for `strstr`, which computes the hash of every sub-string of X and compares with H_c.

3.1.2 Decipher Routine

In our implementation, we selected AES with 256-bit keys as the encryption algorithm. Constants in the conditions are not directly used as keys because of the varying type and length. A key generation function $Key(X)$ is used to produce 256-bit keys. We describe the function in more details in the next section.

We use the LLVM API to place a call immediately before the original conditional code to a separate decipher function that can be dynamically or statically linked to the malware program. Figure 7 illustrates this for a basic block. For a function, the call to the decipher routine is inserted as the first statement in the function body. The $Decipher$ routine takes two arguments. The first is a dynamically computed key $Key(X)$, which is based on the variable X in the condition. The second is the length of the code to be decrypted by the decipher routine. When calling a function that is to be obfuscated, these two arguments are added to the list of arguments for that function. This allows them to be passed not only to decipher routine called in that function body, but also to other obfuscated functions that the function may call. At this stage in the obfuscation scheme, this length is not known because the final generated code size will vary from the intermediate code. We keep a place holder so that the actual value can be placed during binary analysis.

3.1.3 Decryption Key and Markers

The key generation function Key uses a SHA-256 cryptographic hash to generate a fixed length key from varying length data. However, the system would break if we used the same hash function as used in the condition. The reason is that if the key $Key(c) = Hash(c) = H_c$, then the stored hash H_c in the condition can be used to decrypt the code blocks. Therefore, we use $Key(X) = Hash(X|N)$, where N is a nonce. This ensures that the encryption key $K_c = Key(c) \neq H_c$, where c is the constant in the condition. At this stage, the code block to be encrypted is not modified. Immediately following this block, we place the encryption key K_c.

During intermediate code analysis, it is not possible to foresee the exact location of the corresponding code in the resulting binary file. Therefore, we place markers in the code, which are later identified using a binary analyzer in order to perform encryption. The function call to $Decipher$ works as a beginning marker and we place a dummy function call $End\_marker()$ after the encryption key. We use function calls as markers because the LLVM optimization removes other unnecessary instructions from the instruction stream. This type of placement of the key and markers have no abnormal consequences on the execution of the program because it is identified during binary analysis and removed at the final stage of obfuscation.

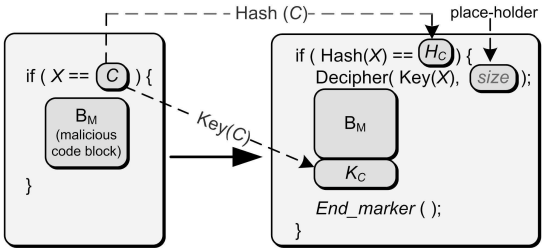


Figure 7. Analysis phase (performed on IR).

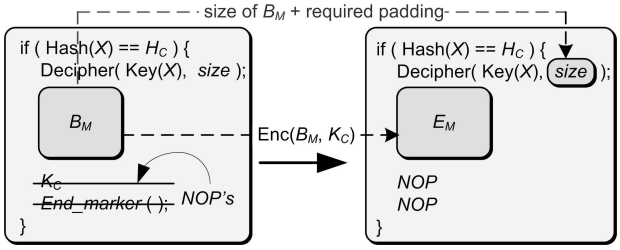


Figure 8. Encryption Phase (performed on binary).

3.2 Encryption Phase

With the help of DynInst, our tool analyzes the ELF binary output by LLC. In order to improve code identification, we ensure that symbol information is intact in the analyzed binary.

Figure 8 illustrates the steps carried out in this phase. At this stage, our tool identifies code blocks needing encryption by searching for calls to the marker functions $Decipher()$ and $End\_marker()$. When such blocks are found, it extracts the encryption key K_c from the code and then removes the key and the call to the End_marker function by replacing them with x86 `NOP` instructions. It then calculates the size of the encrypted block. Since AES is a block cipher, we make the size a multiple of 32 bytes. This can always be done because the place for the key in the code leaves enough `NOP`s at the end of the code block needing encryption. We place the size as the argument to the call to $Decipher$, and then encrypt the block with the key K_c.

The nested conditional code blocks must be managed in a different way. We recursively search for the innermost nested block to encrypt, and perform encryption starting from the innermost one to the outermost one. Since our method of encrypting the code block does not require extra space beyond what is already reserved, our tool does not need to perform code movement in the binary.

3.3 Run-time Decryption Process

The $Decipher$ function performs run-time decryption of the encrypted blocks. Notice that the location of the block that needs to be decrypted is not sent to this function. When the $Decipher$ function is called the return address pushed onto the stack is the start of the encrypted block immediately following the call-site. Using the return address pushed on the stack, the key and the block size, the function decrypts the encrypted block and overwrites it. Once the block has been decrypted, the call to the $Decipher$ function is removed by overwriting it with `NOP` instructions.

The decryption function uses the ability to modify code. Therefore, write protection on the code pages is switched off before the modification and switched back on afterwards.

4 Experimental Evaluation

We used our obfuscation tool on several malicious programs in order to evaluate its ability to hide trigger based behavior. Although we had to select from a very limited number of available malicious programs written for Linux, we chose programs that are representative of different classes of malware for which triggers are useful.

We evaluated our system by determining how many manually identified trigger-based malicious behaviors were automatically and completely obfuscated as conditional code sections by our system. In order to evaluate the resistance to brute force attacks on each obfuscated trigger, we defined three levels of strength depending on the type of the data used in the condition. An obfuscation was considered *strong*, *medium*, or *weak* if its condition incorporated strings, integers, or boolean flags, respectively. Table 1 shows the results of our evaluation on various programs. Notice that almost all trigger based malicious code was successfully obfuscated using our tool. However, for a few specific instances our tool was either able to provide weak obfuscation or not able to provide any obfuscation at all. We consider both of these cases as a failure for our obfuscation to provide any protection. We investigated the reasons behind such cases and describe how the malware program could be modified to take advantage of our obfuscation.

We first tested our tool on the Slapper worm [38]. Although it is a worm, it contains a large set of trigger based behaviors found in bots and backdoor programs. When the worm spreads, it creates a UDP based peer-to-peer network among the infected machines. This entire network of victims can be controlled by sending commands to carry out Distributed Denial of Service attacks. In addition, it installs a backdoor program on the infected machine that provides shell access to the victim machine.

The Slapper worm has two components. The first

Malware	Malicious triggers	Strong	Medium	Weak	None
Slapper worm (P2P Engine)	28	-	28	-	-
Slapper worm (Backdoor)	1	1	-	-	-
BotNET (An IRC Botnet Server)	52	52	-	-	-
passwd rookit	2	2	-	-	-
login rootkit	3	2	-	-	1
top rootkit	2	-	-	-	2
chsh rootkit	4	2	-	2	-

Table 1. Evaluation of our obfuscation scheme on automatically concealing malicious triggers.

(`unlock.c`) contains the worm infection vector and the code necessary to maintain the peer-to-peer network and receive commands. We manually identified 28 malicious triggers in the program that performs various malicious actions depending on the received control commands. These triggers were implemented using a large `switch` construct. Our tool was able to completely obfuscate all malicious actions of these triggers. However, the obfuscations had medium-level strength because the conditions were based on integers received in the UDP packets. The second part of the worm is a backdoor program (`update.c`) that opens a Linux shell when the correct password is provided to it. The program contained only one malicious trigger, which uses the `strcmp` function to check whether the provided password was equal to the hard-coded string "aion1981". Our tool was able to successfully obfuscate the entire code of this trigger (which included a call to `execve`) and removed the password string from the program. In this case, our tool provided strong obfuscation because the condition was based on a string.

We next tested our obfuscation on a generic open source bot program BotNET for Linux, which had minimal command and control support built into it. Since bots typically initiate different malicious activities after receiving commands, we identified the sections in that program that receive commands and considered them as malicious triggers. We manually found 52 triggers in the program, and after our obfuscation was applied, code conditionally executed for all 52 of them were strongly obfuscated. This result represents what we can expect by obfuscating any typical bot program. Usually, all IRC based bot commands send and receive text based commands, making the triggers suitable for strong obfuscation.

We found several rootkit programs [30] for Linux that install user level tools similar to trojan horse programs containing specific logic bombs that provide malicious advantages to attackers, including privileged access to the system. First, we tested the `passwd` rootkit program, which had two manually identifiable malicious triggers inserted into the original. The first trigger enables a user to spawn a privileged shell when a predefined magic string "satori" is inserted as the new password. The second trigger works in a similar manner and activates when the old password is equal to the magic string. Our obfuscation successfully concealed these trigger based code with strong obfuscation.

We next tested on a rootkit version of `login`. The source code `login.c` contained three manually identified malicious triggers. Our obfuscation was able to conceal two with strong obfuscation. Both of these triggers were `strcmp` checks on the user name or password entered by the user. Our obfuscator was not able to protect the third trigger. The reason was that both of the strongly obfuscated code sections increase the value of an integer variable `elite` that was used by the third trigger placed elsewhere in the program. The third trigger used the operator `!=`, making it unsuitable for our obfuscation.

Our next test was on the `top` rootkit. This program contained two malicious triggers, which hide specific program names from the list that a user can view. Although these triggers are implemented using `strcmp`, the actual names of the processes that are to be hidden are read from files. Our obfuscator therefore could not conceal any of these malicious triggers. A malware author could take a different approach to overcome situations like this. By having a trigger that checks for the file containing the process names to start with a hard-coded value, all the other triggers can be contained in a strongly obfuscated code section.

Finally, we tested on a rootkit that is a modified version of the `chsh` shell. Two triggers were manually identified. The first, which checked for a specific username, was strongly obfuscated by our tool. The second trigger used a boolean flag in and therefore was only weakly obfuscated. It is easy for one to overcome this difficulty because by manually modifying the flag to be a string or an integer, stronger obfuscation can be obtained using our tool.

5 Discussion

In this section, we discuss our obfuscation technique to help defenders better understand the possible threats it poses. We discuss how malware authors (attackers) may utilize this technique, and analyze its weaknesses to provide insight into how such a threat can be defeated.

5.1 Strengths

We have discussed earlier in Section 2.4 how our obfuscation impedes state-of-the-art malware analyzers. If the the variable used in the condition has a larger set of possible reasonable values, the obfuscation is stronger against brute force attacks. Since data type information is hard to determine at the binary level, brute-force attacks may have to search larger sets of values than necessary, providing more advantage to the attackers. Equipped with this knowledge, a malware author may modify his programs to take advantage of the strengths rather than naively applying it to the existing programs.

First, a malware author can modify different parts of a program to introduce more candidate conditions. Rather than passing names of resources to system calls, he can query for resources and compare with the names. In addition, certain conditions that use other relational operators such as $<$, $>$ or $\neq$ that are unsuitable for obfuscation may be replaced by $==$. For example, time based triggers that use ranges of days in a month can be replaced with several equality checks on individual days. As another example, a program that exits if a certain resource is not found by using the != operator, can be modified to continue operation if the resource is found using the == operator.

Second, a malware author can increase the size of the concealed code in the malware programs by incorporating more portions of the code under candidate conditions for obfuscation. Bot authors can have an initial handshaking command that encapsulates the bulk of the rest of its activity.

Third, the malware authors can increase the length of inputs to make brute force attacks harder. In our implementation, we use AES encryption with 256-bit keys. If any variable used to generate the key is less than 256-bits in size, then the effective key-length is reduced. Attackers may also avoid low entropy inputs for a given length because that may reduce the search space for dictionary-based attacks. For example, a bot command that uses lower case characters only in a buffer of 10 bytes needs the search space of size 26^{10}, whereas using both upper, lower, and numeric values would increase search space to 62^{10}.

Unlike strings, numeric values usually have a fixed size and may not be increased in length. Checking the hashes of all possible 32-bit integer values may not be a formidable task, but it is time-consuming, especially if a different nonce is used in each condition to make pre-computed hashes not useful. An attacker can utilize some proprietary function $F(x)$ in the code to map a 32-bit integer x to a longer integer value. However, such an approach does not fundamentally increase the search space. It may just increase the difficulty for the defenders in automatically computing the hashes because the equivalent computation of F has to be extracted from the code and applied for each possible 32-bit value before applying the hash function.

5.2 Weaknesses

In its current form, one of the main weaknesses of our obfuscation is the limited types of conditions on which it can be applied to. Although triggers found in the programs that we experimented with were mostly equality tests, there can be many trigger conditions in a malware that checks ranges of values. If the range is large, it may not be possible to represent them as several equality checks as we have mentioned earlier, making the obfuscation inapplicable.

The encryption strength depends on the variable that is used. The full strength of having 2^{256} possible keys for AES is not utilized particularly in the case of numeric data, which are 32-bit or 64-bit integers in current systems. Obfuscations involving string inputs are likely to be more resistant to analysis. Therefore, a subclass of malware, especially bots or backdoors, are likely to be the most beneficial from this approach because the inputs required for the malicious triggers can be selected by the malware authors.

Another weakness is that trigger-based behavior may not just depend on data that are input using system calls, but also status results returned from the calls. Most system calls have a small set of possible return values, usually indicating a success or some form of error. As a result, the number of values to check by a brute force attack may be reduced even further for such conditions.

Possible ways to defeat: If the proposed obfuscation is successfully applied, existing malware analysis techniques may not be able to extract the behavior that is concealed unless the conditions in the triggers are satisfied. Yet, we suggest several techniques to defeat our obfuscation.

First, analyzers may be equipped with decryptors that reduce the search space of keys by taking the input domain into account. Once an obfuscated condition is detected, the variable used in it may be traced back to its source using existing methods. If it is the result or an argument receiving data from a system call, the corresponding specification may be used to find the reasonable set of values that it may take. For example, if the source is set by a calling system call that returns the current system date (such as `gettimeofday`), the set of all possible values representing valid dates may be used, significantly reducing the search space. If, however, the source is input data that cannot be characterized, brute forcing may become infeasible.

Another approach can be to move more towards input-aware analysis. Rather than capturing binaries only, collection mechanisms should capture interaction of the binary with its environment if possible. In case of bots, having related network traces can provide information about the inputs required to break the obfuscation. Existing honey-

pots already have the capability to capture network activity. Recording system interaction can provide more information about the inputs required by the binary.

Malware detection: Although the obfuscation may prove powerful against malware analyzers, its use may have an upside in malware detection. The existence of hash functions and encryption routines together with a noticeable number of conditions utilizing them may indicate that an unknown binary is a malware. However, our proposed obfuscation allows more layers of other general obfuscation schemes to be applied on top of it. For example, the binary resulting from our system may be packed with executable protectors [34], which a large fraction of malware already do today. The use of protector tools alone are not usually an indication that the program is a malware because these tools are usually created for protecting legitimate programs. Removing such obfuscation layers require unpacking techniques [21, 33] that are mostly used prior to analysis of suspicious code because of their run-time cost. The end result is that detecting such obfuscated malware is not any easier than detecting existing ones.

6 Related Work

The problem of discovering trigger-based malicious behaviors has been addressed by recent research. Some techniques have used symbolic execution to derive predicates leading to specific execution paths. In order to identify time-bombs in code, [13] varies time in a virtual machine and uses symbolic execution to identify predicates or conditions in a program that depend on time. Another symbolic execution based method of detecting trigger based behavior is presented in [5]. Brumley et al.'s Bitscope [4] uses static analysis and symbolic execution to understand behavior of malware binaries and is capable of identifying trigger-based behavior as well. Our obfuscation makes it hard for symbolic constraints containing cryptographic one-way hash functions to be solved.

Approaches have been proposed that identify trigger-based behavior by exploring paths during execution. Moser et al.'s multi-path exploration approach [28] was the first such approach. The technique can comprehensively discover almost all conditional code in a malware with sufficient execution time. The system uses QEMU and dynamic tainting to identify conditions and construct path constraints that depend on inputs coming from interesting system calls. Once a conditional branch is reached, the approach attempts execution on both of the branches after consistently changing memory variables by solving the constraints. Another approach is presented in [42] that forces execution along different paths disregarding consistent memory updates. The approach has been shown to be useful for rootkits written as Windows kernel drivers.

Techniques that can impede analyzers capable of identifying trigger-based behavior need to conceal conditions and the code blocks that are used to implement these behaviors. An example of obfuscated conditional branches in malware was seen in the early 90s in the Cheeba [19] virus, which searched for a specific file by comparing the hash of the name of each file with a hard-coded hash of the file name being searched. In the literature, the idea of using environment generated keys for encryption was introduced in [32]. The work on secure triggers [18] considers a whitehat scenario and presents the principles of constructing protocols for software developers to have inputs coming into a program to decrypt parts of the code. In the malware scenario, the research idea of using environment generated keys for encryption was presented as the Bradley virus [17]. However, such techniques have not become a practical threat because identification of such keys and incorporation of the encryption technique requires a malware to be manually designed and implemented around this obfuscation. Our work shows that encrypting parts of the malware code using keys generated from inputs can be automatically applied on existing malware without any human effort, showing its efficacy as a wide-spread threat.

The use of polymorphic engines in viruses is one of the earliest [36] obfuscation techniques used in malware to evade detection. Over the years, various methods of polymorphic and metamorphic techniques have appeared in the wild [39]. In order to detect polymorphic malware, antivirus software use emulation or create signatures to detect the polymorphic decryptors. In [9], obfuscation techniques such as garbage insertion, code transposition, register reassignment, and instruction substitution were shown to successfully disrupt detection of several commercial antivirus tools.

Besides malware detection approaches [10, 22, 23], recent research has focused on creating techniques that automate malware analysis. These systems automatically provide comprehensive information about the behavior, runtime actions, capabilities, and controlling mechanisms of malware samples with little human effort. A variety of obfuscation techniques [11,12,25,43] have been presented that can impede such analyzers that are based on static analysis approach. Executable protectors and packers [34] are widely used by malware authors to make reverse engineering or analysis of their code very hard. As with polymorphic code, packers obfuscate the code by encrypting or compressing the binary and adding an unpacking routine, which reverses the operation during execution. Tools [21, 26, 33] have been presented that are able to unpack a large fraction of such programs to aid static analysis. However, by utilizing our obfuscation before packing, malware authors are capable of concealing code implementing triggered behavior from static analyzers even after unpacking is performed.

Dynamic analysis approach has been more attractive for automated malware behavior analyzers. This is because performing analysis of code that is executing overcomes obfuscations that impede static analysis, including packed code. Since most dynamic analysis of malware involves debuggers [15], safe virtual machine execution environments or emulators, malware programs use various anti-debugging [8] and anti-analysis techniques to detect side-effects in the execution environment and evade analysis. There has been research on stealth analysis frameworks such as Cobra [40], which places stealth hooks in the code to aid analysis while remaining hidden from the executing malware. In order to automate analysis, dynamic tools such as CWSandbox [6], TTAnalyze [2] or the Norman Sandbox [29] automatically record the actions performed by an executing malware. However, since such tools can only view a single execution path, trigger-based behavior may be missed. These tools have been superseded by the recent approaches that can identify and extract trigger-based behavior, which we have presented earlier in this section.

7 Conclusion

We have designed an obfuscation scheme that can be automatically applied on malware programs in order to conceal trigger based malicious behavior from state-of-the-art malware analyzers. We have shown that if a malware author uses our approach, various existing malware analysis approaches can be defeated. Furthermore, if properly used, this obfuscation can provide strong concealment of malicious activity from any possible analysis approach that is oblivious of inputs. We have implemented a Linux based compiler level tool that takes a source program and automatically produces an obfuscated binary. Using this tool we have experimentally shown that our obfuscation scheme is capable of concealing a large fraction of malicious triggers that are found in several unmodified malware source programs representing various classes of malware. Finally, we have provided insight into the strengths and weaknesses of our obfuscation technique and possible ways to defeat it.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCR-0133629, CNS-0627477, and CNS-0716570, and by the U.S. Army Research Office under Grant W911NF0610042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation and the U.S. Army Research Office.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, & Tools*. Addison Wesley, 2006.

[2] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of the 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[3] R. Benzmller and T. Urbanski. G DATA malware report 2006. G Data Software AG, 2006.

[4] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, S. D, and H. Yin. Bitscope: Automatically dissecting malicious binaries. In *CMU-CS-07-133*, 2007.

[5] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Towards automatically identifying triggerbased behavior in malware using symbolic execution and binary analysis. *Technical Report CMU-CS-07-105, Carnegie Mellon University*, 2007.

[6] C. Willems. CWSandbox: Automatic Behaviour Analysis of Malware. http://www.cwsandbox.org/, 2006.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engleri. EXE: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2006.

[8] P. Cerven. *Crackproof Your Software: Protect Your Software Against Crackers*. 2002.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the Usenix Security Symposium*, 2003.

[10] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. In *Technical Report 148, Department of Computer Sciences, The University of Auckland*, July 1997.

[12] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL98)*, January 1998.

[13] J. Crandall, G. Wassermann, D. Oliveira, Z. Su, F. Wu, and F. Chong. Temporal search: Detecting hidden malware timebombs with virtualmachines. In *Proceedings of the Conference on Architectural Support for Programming Languages and OS*, 2006.

[14] D. Dagon. Botnet detection and response: The network is the infection. In *Proceedings of the OARC Workshop*, 2005.

[15] Data Rescue. IDA Pro Disassembler and Debugger. http://www.datarescue.com/idabase/index.htm.

[16] J. Ferrante, K. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, volume 9, pages 319–349, 1987.

[17] E. Filiol. Strong cryptography armoured computer viruses forbidding code analysis. In *Proceedings of the 14th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2005.

[18] A. Futoransky, E. Kargieman, C. Sarraute, and A. Waissbein. Foundations and applications for secure triggers. In *ACM Transactions of Information Systems Security*, volume 9, 2006.

[19] D. Gryaznov. An analysis of cheeba. In *Proceedings of the Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 1992.

[20] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the Scalable High Performance Computing Conference*, 1994.

[21] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM 2007)*, 2007.

[22] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of the Usenix Security Symposium*, 2006.

[23] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the Annual Computer Security Application Conference (ACSAC)*, 2004.

[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[25] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.

[26] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[27] D. Moore, C. Shannon, , and J. Brown. Code-red: A case study on the spread and victims of an internet worm. In *Proceecings of the 2nd ACM Internet Measurement Workshop*, 2002.

[28] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium of Security and Privacy*, 2007.

[29] Norman Sandbox Information Center. http://www.norman.com/microsites/nsic/, 2006.

[30] Packet Storm Security. http://www.packetstormsecurity.org/.

[31] S. Pearce. Viral polymorphism. VX Heavens, 2003.

[32] S. Riordan and B. Schneier. Environmental key generation towards clueless agents. In *Mobile Agents and Security*, 1998.

[33] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[34] Silicon Realms. Armadillo/software passport professional. http://www.siliconrealms.com/index.html.

[35] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Transactions on Software Engineering and Methodology*, 10(2):209–254, 2001.

[36] F. Skulason. *1260-The Variable Virus*. Virus Bulletin, 1990.

[37] Symantec - Virus Database. Keylogger.stawin. http://www.symantec.com/security_response /writeup.jsp?docid=2004-012915-2315-99.

[38] Symantec - Virus Database. Linux.slapper.worm. http://securityresponse.symantec.com/avcenter/ security/Content/2002.09.13.html.

[39] P. Szor. *The Art of Computer Virus Research and Defense*. Symatec Press, 2005.

[40] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[41] I. V.Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proceedings of the Usenix Security Symposium*, 2007.

[42] J. Wilhelm and T. cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)*, 2007.

[43] G. Wroblewski. Generalmethod of program code obfuscation. PhD thesis, Wroclaw University of Technology, 2002.

[44] A. Young and M. Yung. Cryptovirology: Extortion based security threats and countermeasures. In *Proceedings of the IEEE Symposium of Security and Privacy*, 1996.