DEOBFUSCATION OF PACKED AND VIRTUALIZATION-OBFUSCATION PROTECTED BINARIES

by

Kevin Patrick Coogan

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2011

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Kevin Patrick Coogan
entitled Deobfuscation of Packed and Virtualization-Obfuscation Protected Binaries
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Saumya Debray — Date: 16 June 2011

Peter Downey — Date: 16 June 2011

Christian Collberg — Date: 16 June 2011

John Hartman — Date: 16 June 2011

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Saumya Debray — Date: 16 June 2011

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

SIGNED: Kevin Patrick Coogan

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Saumay Debray for being exactly the type of advisor that I needed. It is no easy task to know when to help a struggling student, and when to back off and let him learn on his own. He has shown extraordinary skill and wisdom in this respect, and I am forever grateful. I would also like to thank the rest of my committee–Pete Downey, Christian Collberg, and John Hartman–for sharing their experiences with me. It has been an honor and a privilege to work with each of them. A special thank you to Lester McCann, who has taught me everything I know (but not everything he knows) about teaching.

I would also like to acknowledge the contributions of my colleagues during my time at Arizona. Specifically, thank you to Tasneem Kaochar and Gregg Townsend for their efforts in implementing the value-set analysis work described in Section 3.5 of this dissertation. Also, thank you to my fellow Ph.D. student Gen Lu for his work on handling pointer parameters described in Section 4.2.1. Finally, a special thanks to Keith Fligg, who was the best "sounding board" I have ever met. I would not have succeeded without all your help.

I would like to acknowledge the tireless work of the Computer Science Department lab staff for all their efforts to keep me working, as well as the efforts of the staff in the academic and financial offices for keeping me on track all these years.

Grad school would probably not be survivable without some down time for fun and relaxation. I would like to thank all of my friends and fellow grad students who have made this a good life, not just a good job. Specifically, Igor Crk, Varun Khare, Kyri Pavlou, and Joe Roback. I cannot imagine a better group to waste time with.

Also, thank you to my parents, who never once wavered in their support of me when I decided to go back to school to pursue my doctoral degree.

Last, but not least, a very special thank you to Nassim Mafi for making all of my hard work meaningful. I love you azizam, always.

DEDICATION

For Nassim, with love and gratitude.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Code obfuscation techniques are increasingly being used in software for such reasons as protecting trade secret algorithms from competitors and deterring license tampering by those wishing to use the software for free. However, these techniques have also grown in popularity in less legitimate areas, such as protecting malware from detection and reverse engineering. This work examines two such techniques – packing and virtualization-obfuscation – and presents new behavioral approaches to analysis that may be relevant to security analysts whose job it is to defend against malicious code. These approaches are robust against variations in obfuscation algorithms, such as changing encryption keys or virtual instruction byte code.

Packing refers to the process of encrypting or compressing an executable file. This process "scrambles" the bytes of the executable so that byte-signature matching algorithms commonly used by anti-virus programs are ineffective. Standard static analysis techniques are similarly ineffective since the actual byte code of the program is hidden until after the program is executed. Dynamic analysis approaches exist, but are vulnerable to dynamic defenses. We detail a static analysis technique that starts by identifying the code used to "unpack" the executable, then uses this unpacker to generate the unpacked code in a form suitable for static analysis. Results show we are able to correctly unpack several encrypted and compressed malware, while still handling several dynamic defenses.

Virtualization-obfuscation is a technique that translates the original program into virtual instructions, then builds a customized virtual machine for these instructions. As with packing, the byte-signature of the original program is destroyed. Furthermore, static analysis of the obfuscated program reveals only the structure of the virtual machine, and dynamic analysis produces a dynamic trace where original program instructions are intermixed, and often indistinguishable from, virtual

machine instructions. We present a dynamic analysis approach whereby all instructions that affect the external behavior of the program are identified, thus building an approximation of the original program that is observationally equivalent. We achieve good results at both identifying instructions from the original program, as well as eliminating instructions known to be part of the virtual machine.

CHAPTER 1

INTRODUCTION

In the context of computer security, the term *obfuscation* refers to the practice of making programs more difficult to understand or reverse engineer (see Collberg and Thomborson (2002), Collberg et al. (1997)). Code obfuscation can refer to many different techniques including (but not limited to) encryption, compression, rewriting to equivalent instructions, and control flow flattening, just to name a few. These techniques have several legitimate applications such as protection of copyrighted software algorithms from being reverse engineered by the competition, or by individuals trying to use the software for free. However, they have also found popularity among authors of malicious programs (e.g., viruses, worms) for protection of their code against detection or reverse engineering. The work presented in this dissertation centers on techniques for deobfuscating programs that have been protected using two specific obfuscation techniques – packing and virtualization-obfuscation. While this work is equally applicable to any program employing such techniques, no matter its intent, the use of such techniques in malware is a strong motivating factor and we spend considerable time discussing the work in this context.

The dissertation is organized as follows. Chapter 1 gives a high level overview of the motivation, background, and our basic approach and conclusions, and is meant to stand alone as a brief summary of the entire work. Chapter 2 gives a detailed look at relevant background information, including a discussion of how the packing and virtualization-obfuscation techniques work, and a review of any work by other authors that we build on. Chapter 3 details the static analysis work with regard to packing obfuscation. Chapter 4 describes the dynamic analysis used to identify all instructions in a dynamic trace relevant to the behavior of the instance of execution that created the trace. Chapter 5 extends the work of Chapter 4 and describes in

detail the analysis done to identify conditional control flow statements in the same trace so that a more general understanding of the behavior of the original program can be achieved. Finally, Chapter 6 summarizes the findings that result from this work.

1.1 Motivation

The term *malware* refers generally to any of a number of types of computer programs that are designed to take control of a computer system without the owner's consent. Early on, such malicious code was straightforward, and typically created either as an academic exercise, or by programmers attempting to impress one another. This malware typically consisted of some code whose purpose was to thwart any protections and infect the machine. It also typically contained a payload – some code to be executed once the system was infected. In these early examples, the payload was often as simple as printing a message giving credit for the program to its author, or possibly copying itself to other files on the system in the hopes of spreading to more machines.

Current malware is often written and distributed by large organizations that wish to illegally use the infected machines, for example, as servers for their content, or as mail servers for sending out spam. While the source of these malicious programs has changed, their structure remains very similar. There is still some portion of code dedicated to infection (i.e., defeating protections), and a payload that performs some task (e.g., sending spam e-mails). However, modern malware also typically contains code whose purpose is to prevent detection or delay reverse engineering.

In the case of these protections, the authors of the malware have a built-in advantage over the security analysts who are trying to stop them. Modern malware need only be active for a short time to be effective. The end goal is to take control of as many machines as possible. Once a new piece of malware is introduced, there will naturally be a certain amount of time that elapses before it is detected, during which time the code is taking control of as many machines as possible.

Even after detection, the malware often has to be understood and analyzed before counter-measures can be developed. In the past, it was enough to create a byte-signature for the malware that could be included in anti-virus software so that the malware could be effectively detected from that point forward. This technique is no longer adequate, however. Modern malware often use a variety of techniques that allow each instance (i.e., each copy) of a malicious program to carry a different byte signature. Thus, the code must be reverse engineered, and its behavior understood before defenses can be created.

To make matters worse for the security analysts, much of the analysis of these malware are still done by hand. This process can be very difficult, tedious, and error prone. After this process is completed, the defenses must be propagated to anti-virus software, and techniques for cleaning already infected machines must be followed. Two examples suggest how time-critical this work can be. First, in 2003, the *SQLSlammer* virus was released and infected 75,000 machines in its first 10 minutes in the wild (see Moore et al. (2003)). Second, around 2008, the *Conficker* worm was released. A year later, security analysts admitted that they still didn't understand everything about how the program worked. All the while, it continued infecting machines, with estimates of the number of infections as high as 8.9 million (see Lawton (2009)).

Clearly, there is a need for tools and techniques that security analysts can use to automate part or all of this process. The work presented in this dissertation attempts to handle two specific cases of protections known to be used by malicious code. As stated, these techniques are equally applicable to other protected programs, regardless of their usage or author's intent.

1.2 Basic Approach

In the past, anti-virus software identified malware by their byte-signatures–patterns of bytes that appear in the executable code of the virus, but not in other files. Deriving a byte-signature for a particular file could be tricky, but once found was a

reliable means of identifying the existence of the malware. With the emergence of new obfuscation techniques (e.g., see Song et al. (2007)), the fundamental assumption behind this approach is no longer valid. Consider the case of using simple key encryption to pack an executable file. A key is chosen, and applied to each byte of the program (or the part to be encrypted), typically through some operation such as an xor operation. If any of the bytes that are changed were used in the byte-signature, then the encrypted form of the file will not be recognized by the anti-virus software. The naive approach would be to add the byte-signature of the encrypted version of the file to the anti-virus software. However, this does not fix the problem, since the encryption key could easily be changed to produce yet another different version of the file. In fact, each possible encryption key will yield a different version of the file. Combine this with all the various ways to apply that key to the bytes of the executable (e.g., xor, add, sub), and the approach becomes impractical.

The problem exists for virtualization-obfuscation protected code as well. The first step of virtualizing a program is to convert the instructions of the original program into virtual instructions to be read by a custom virtual machine. There are an almost limitless number of possible ways to assign byte code values to each of these instructions, thus an almost limitless number of byte-signatures would need to be derived and stored. Again, the approach becomes impractical.

As a result, we propose a behavior based approach for looking at obfuscated malware. As an example, we observe that while changing the encryption key produces radically different byte code for the same executable, the behavior of packing and unpacking stays pretty much the same. The bytes of the file are altered using some algorithm, then restored by reversing the algorithm just before execution. Ideally, there would be a single, general analysis technique that could not only handle all known protections, but also anticipate any future techniques that may arise. While common sense suggests that such an analysis will never exist, looking for more general approaches that are based on behavior rather than byte-signatures is a worthy goal. In time, this search could, hopefully lead to a collection of analysis tools that are flexible enough to be applied to a wide variety of cases, and which will require

little to no modification to handle new protection schemes. This work is an early step in this direction.

We begin by looking at the obfuscation technique known as "packing," leveraging the known behaviors and characteristics of packed programs to undo their protection. Despite promising early results, this work reveals some significant shortcomings of a static approach in the context of malware analysis. Next, we apply dynamic analysis to the obfuscation technique known as "virtualization-obfuscation," or sometimes "emulation" or just "virtualization." We are able to identify the behavior of a particular instance of execution by identifying all system calls made by the program, as well as all instructions in the trace that contribute to the values of the parameters to those system calls. This work shows great promise because it is fairly robust to the types and degree of virtualization that are employed. However, it can only identify data dependencies to the system call parameters, and thus can only say something about the behavior of the original program on one specific set of inputs. Finally, we broaden the scope of the virtualization analysis by applying additional dynamic analyses to identify known conditional control flow statements. This conditional control flow is the first step towards rebuilding the original, unprotected program that can handle multiple inputs.

1.2.1 Packing

The obfuscation technique known as *packing* refers generally to any process where the bytes of an executable file have been altered by some routine, and must necessarily be returned to their original form before execution. Common examples include *encryption*, where the bytes of the file (or some portion of the file) are encrypted, e.g., with an encryption key, and *compression* where the file is made smaller on disk through some algorithm. In all such cases, the alteration of the bytes means that the instructions represented by those bytes are now obfuscated and no longer executable. The side effect of these processes is that any byte-signature contained in the executable form of the program is now completely destroyed, thus rendering the byte-signature matching algorithms of traditional anti-virus programs ineffective.

Of course, the bytes must be returned to executable form before the program can be of any use. This task is handled by the *unpacker* routine. The unpacker routine is not part of the original program, but is a routine added to the code during packing. In its simplest form, it iterates over the packed bytes, returning them to their original form, then transfers program control to the newly restored code. The file is altered so that when the file is executed, the unpacker runs first, unpacking the code into memory. The result of this approach is that the unpacked code never exists on disk, only in memory after the file has started execution. Straightforward static analysis will try to disassemble the altered bytes and build a control flow graph. The result will be no disassembly since the bytes do not translate into instructions, or worse, a wrong disassembly that has confused the altered bytes with actual instructions. For this reason, previous attempts at defeating packing (see Martignoni et al. (2007); Kang et al. (2007)) typically rely on a dynamic approach that identifies when the unpacked code exists in memory, then dumping the contents of memory and disassembling those contents.

The use of dynamic analysis makes these attempts vulnerable to dynamic defenses. For example, some malware are known to check system values and only execute their payload under certain conditions. It is possible that a malware unpacker routine will test for the current date, and only unpack the code if it is Friday the 13^{th}. If the dynamic analysis of this code is not run on the correct date, then the unpacked payload will never be revealed. As another example, it is possible that the malware may unpack a small portion of the code, then execute it, then unpack another portion, and so on. In this case, previous work will detect the unpacking of the first portion, and correctly disassemble it, but may not be able to identify unpacked code that would be executed later.

Basic Approach

Our approach (see Coogan et al. (2009)), discussed in detail in Chapter 3, uses static analysis techniques, but leverages the characteristics of packed code to correctly disassemble the executable. It makes use of several observations about packed code.

First, that the unpacker routine must be present, that it must execute before the packed code, and that the act of unpacking then executing the original bytes is an example of self-modifying code.

The first step of the analysis is to disassemble the packed binary as much as possible. Since much of the code is packed, there will be errors in the disassembly. However, since the unpacker routine must execute before the unpacked code, a recursive disassembler should correctly identify the unpacker code. A control flow graph (CFG) can be built that will also have errors, but which will have correct execution path information for the unpacker routine.

The next step is to identify all of the instructions in the CFG that may potentially write to memory, and what memory locations they write to. To accomplish this, a value-set analysis, proposed by Balakrishnan (2007) in his Ph.D. dissertation, is implemented for the system. This analysis is safe, meaning that it is guaranteed to include memory locations that are written to, but possibly at the expense of adding some memory locations that are not written to.

Next, a list of potential transition points is generated. A *transition point* is the point in the code where execution moves from an instruction that has not been modified (i.e., unpacked) to an instruction that has been unpacked. In other words, the point where execution moves from the unpacker to the unpacked code. A *potential* transition point, then, is any place in the code where this might be happening, or where execution of self-modified code has begun. The list is generated by intersecting the list of modified memory locations, produced by the safe value-set analysis, with the information from the control flow graph that indicate these locations may be executed after being modified.

For each potential transition point, a backward static slicing is performed that identifies all the instructions that may have affected the value of the bytes to be executed at that point. Then those instructions are emulated in a safe environment. If the potential transition point is a real transition point (i.e., the transition from the unpacker to the unpacked code), then the instructions identified by the slicing algorithm will be the unpacker routine.

As a final step, conditional logic in the unpacker routine is identified and analyzed. If the logic controls whether or not execution reaches the transition point, then it is labeled a dynamic defense and can be neutralized.

Results and Discussion

Initial tests on packed malware indicate good success with this approach. We were able to correctly unpack real world encrypted malware with no prior knowledge of the encryption key used. Similarly, we were able to correctly unpack real world malware compressed with publicly available tools such as UPX, also with no prior knowledge of the algorithm used.

The malware samples that we considered for testing did not employ dynamic defenses prior to the execution of the unpacker routine (interestingly, malware that use dynamic defenses after unpacking are commonplace). To test the analysis, we modified existing test files to include several variations of dynamic defenses prior to the execution of the unpacked code. We correctly identified the dynamic defenses in all cases, and received results identical to those for the unmodified versions.

Despite these good results, developing the analysis tool revealed some serious shortcomings of the static approach. Most notable, was the reliance on the precision of the value-set analysis. The less precise the analysis, the more possible transition points are identified. In the worst case, nearly all locations in the code could be identified as potential transition points. Since each point results in a slicing analysis, execution of the analysis could easily become impractical. Since the problem of pointer analysis is known to be undecidable in the general case (see Landi (1992), Ramalingam (1994)), and since we must assume that the authors of malware will know the details of our analysis, and can try to defeat it by intentionally including arbitrary amounts of pointer arithmetic, we cannot hope to create a practical solution to this problem.

In addition, the reliance on accurate disassembly of the unpacker is also a serious concern. This was not an issue in our specific test cases. However, these cases did not employ significant anti-disassembly techniques. Many packers (e.g.,

ASPack Software (2009)) claim the ability to obfuscate the entry point to the code, and otherwise disrupt disassembly. Finally, the emergence of new techniques, such as virtualization-obfuscation discussed in the next section, demonstrate how simply examining the code of an executable, even with a perfect disassembly, may not be enough to understand or reverse engineer the code. As a result, we shift our focus to dynamic techniques for deobfuscation.

1.2.2 Virtualization-Obfuscation

Virtualization-obfuscation is technique whereby instructions in a program are virtualized, and a custom virtual machine is inserted to interpret these instructions. While this technique has legitimate uses in such areas as the protection of proprietary algorithms (see Collberg and Thomborson (2002)), it has also been used by malware authors to protect their programs from reverse engineering. The approach works by translating the instructions of an original binary executable into virtual instructions. A customized virtual machine that can interpret those instructions is then created and inserted into the original code, along with the implementations of the virtual instructions. Execution of the program is typically accomplished by the virtual machine constantly fetching and executing the appropriate virtual instructions, for example through the use of a dispatch routine.

As with packing, virtualization-obfuscation also changes the bytes of the original program so that byte-signature matching algorithms are effectively useless. The translation of instructions into virtual instructions not only introduces new instructions into the code altering the byte-signature somewhat, but since the virtual instructions are fetched by the dispatch routine, they can be stored in memory in any location. This structure makes it possible for multiple different instances of an executable using the same virtual machine, and same virtual instruction implementations. In addition, multiple virtual machines may be used, with almost limitless different mappings of byte code to instructions. Additionally, there are multiple ways to implement different instructions, for example using a loop of `add` instructions to accomplish a multiply. All of this leads to near limitless possible

obfuscations of a single executable.

This technique is also very effective against standard static and dynamic analysis approaches. Basic static analysis will produce a control flow graph that shows the structure of the virtual machine. A dynamic trace of a particular instance of execution will be a mixture of instructions doing the work of the original program, and instructions doing the work of the virtual machine. Depending on the level of complexity in the implementation of the virtual machine or machines, it may be impossible to determine which instructions are which.

Current approaches (see Falliere (2009); Sharif et al. (2009); Rolles (2009)) work by making some assumptions about the structure of the virtual machine, then using this information to identify where in memory the virtual instruction implementations are stored. These instructions can then be disassembled and organized into their original form. While this technique works well for those virtual machines that fit the assumptions, they may not work as well for more complex implementations. For example, programs protected with multiple virtual machines would require that each one be identified and its instructions captured. Furthermore, it is possible that virtual machines could be nested, and there is no indication that these methods could recognize that one VM's virtual instructions are the implementation of another virtual machine.

Basic Approach

Based on the previous discussion of the limitations of static analysis, and on the nature of the problem, we settled on a dynamic solution to the problem. This solution is based on several fundamental observations – that any observable program behavior will be the result of interaction with the system (in the case of Windows operating systems, this implies use of the Windows system calls), that program behavior for a particular instance of execution can be defined by what system calls are made and what values are passed to these calls as arguments, and, finally, that in order to understand the program behavior beyond a single execution trace we must understand something about its conditional control flow. The first two of these

observations lead to an analysis tool that can take a dynamic trace of an instance of execution, and identify all the instructions relevant to the execution behavior. The last observation leads to an extension of this tool that identifies which control flow statements are conditionally dependent.

The identification of the relevant call and parameter instructions (see Coogan et al. (2011)) is straight forward, and is discussed in detail in Chapter 4. The first step is to identify system calls in the dynamic trace, and from them, the parameters that are needed. Next, the tool identifies all instructions from the trace that contribute the value of those parameters. This is done using a modified use-def chain calculation. Normally, we associate U-D chains with static analysis. Here, we treat each instance of an instruction as a unique entity, and identify the instance of an instruction that defines the parameter. To understand the distinction between an instruction and an instance of the instruction, consider the dynamic trace generated by executing a simple loop. While each pass through the loop may execute the same instruction multiple times, each pass is a different *instance* of execution. That is, each time an instruction executes is treated as a different event, with possibly different operand values. We identify the instances of instructions that define each of the values used in that instruction, and continue tracing back through the trace until all values are defined, or until the trace ends.

The identification of the relevant conditional control flow instructions is slightly more complicated, and is discussed in detail in Chapter 5. Due to the nature of the virtualization-obfuscation, any control flow instruction may be used to implement conditional logic. To make matters worse, the virtual machine dispatch routine may use the same control flow instruction for all dispatches, whether they are conditional or not. To identify the relevant instructions, we must know how the target addresses of all control flow statements are calculated. This is more difficult than finding the relevant parameter instructions, since the dynamic U-D chain will only tell us what instructions are used, not how the calculation is done.

To solve this problem, an equational reasoning system was designed in-house (see Coogan and Debray (2011)) specifically for x86 assembly code. This tool allows for

a simplified expression to be created for any variable at any point in the program. In the context of conditional control flow, an expression can be generated for each control flow instruction target address. The terms of the expression can be examined one by one to see if any have a conditional component, and thus, decide if the control flow statement itself is conditionally dependent.

Results and Discussion

The evaluation of the dynamic analysis deobfuscation tool proved to be an interesting problem in and of itself. The primary difficulty centers on the problem of quantifying the similarity between two dynamic traces – that of the obfuscated code, and the approximation of a theoretical trace of the unprotected program. In the end, we settled on a sequence matching approach that measure two different values. First, we calculate the number of instructions from the original program that appear in our approximation. The percentage of original program instructions that are correctly identified is labeled the "relevant percentage." Next, we calculate the number of instructions added by the virtualization-obfuscation process that we successfully identify and eliminate. The percentage of virtual machine instructions that are correctly identified and excluded from the approximation is labeled the "obfuscation percentage."

The analysis was evaluated on two different, commercially available virtualization tools, VMProtect and CodeVirtualizer. Relevant percentage numbers indicate the analysis consistently identifies 50-70% of instructions from the original program, while eliminating approximately 90% of those instructions associated with the virtual machine.

1.2.3 Conclusions

A constant "arms race" of protection and detection techniques between malware authors and security analysts has led to advanced techniques being used by malware authors to propagate this malicious code. These techniques require security analysts

to develop more advanced approaches than those that have been used in the past. It is no longer acceptable to identify individual instances of malware, since any one malicious program may be used to generate an almost unlimited number of variations.

New, more general, and more theoretically sound approaches are needed. There is already work being done that thinks of malware in terms of its behavior, and not in terms of the actual bytes or instructions that are used to implement that behavior.

Lee and Mody (2006) proposes an automated behavior-based classification system based on distance measure and machine learning. Kwon and Su (2010) proposes a system that characterizes high-level object-accessing patterns in malware as regular expressions. Bayer et al. (2009) use taint analysis combined with a dynamic trace of system calls to identify how system information is used in calls or conditional statements. They use this information to develop general behavioral models of malware that are independent of the low level details of the actual execution trace, and that also scale well to input sizes of tens to hundreds of thousands of malware files.

This dissertation presents work that continues along this behavior path. It takes a general approach to defeating two popular protection schemes used by malware today. In the case of packed malicious code, it uses the behavior of the unpacker to identify the instructions it needs to unpack the code, regardless of the algorithm. In the case of virtualization-obfuscation, it identifies the behavior of the original program through its interaction with its environment, and captures the instructions that logically must be part of the program.

CHAPTER 2

BACKGROUND AND RELATED WORK

Obfuscation techniques can be used to protect computer code and its contents from outsiders. (e.g., see Collberg et al. (1997), Wroblewski (2002)). In many cases, such techniques are legitimate, since they protect proprietary algorithms from competitors, or protect licensing software from tampering by users trying to use the software for free. However, obfuscation techniques can also be used to protect malicious code whose purpose is to take control of computer systems without the consent of its owner. These techniques, correctly applied, are capable of defeating detection attempts, thus increasing the chances of successfully infecting a machine. Successful malware can then spread quickly to other machines and repeat the process. Hence, in the context of malware analysis, there is a legitimate need to understand obfuscation techniques, and to develop automated means to defeat them.

This dissertation examines two such obfuscation techniques that are common among malware – packing and virtualization-obfuscation. This chapter begins with a discussion of obfuscation in general, then gives some background on each of these approaches, and the issues that must be overcome to deal with them. Chapter 3 will detail our approach to handling packing, and Chapters 4 and 5 will detail our approach to handling virtualization-obfuscation.

2.1 Obfuscation and Deobfuscation

Code obfuscation in general refers to a class of techniques used to hide or protect information about a computer program, either in source code or executable form. Early work by Collberg et al. (1997) discusses possible legitimate uses of such techniques, such as protection of proprietary algorithms from theft by competitors. It also proposes a taxonomy of obfuscation techniques that, at its highest level, includes

layout obfuscations such as reordering or renaming of functions, data obfuscations such as splitting variables into multiple parts and converting static data to function calls, control obfuscations such as method in-lining and loop unrolling, and preventative transformations that seek to exploit weaknesses in the specific tools of reverse engineering. The work of Wroblewski (2002) expands on this work with a general algorithm for code obfuscation at the machine code level that breaks the process down into four distinct actions–reordering of program instructions, reordering of program blocks, exchange of equivalent fragments, and insertion of additional code.

Multiple applications of obfuscation have been studied in the literature. The problem of Java byte code obfuscation has been studied by Collberg and Thomborson (2002) and Low (1998). Preda and Giacobazzi (2005) looked at obfuscation as a means of protecting against malicious behavior, Linn and Debray (2003) examined the use of obfuscation as a means of deterring static analysis, and Sharif et al. (2008) used obfuscation techniques to impede malware analysis.

Our work deals specifically with deobfuscation, or removing the techniques discussed above. Udupa et al. (2005) studied the problem of reverse engineering obfuscated code in general. They used various static analysis techniques such as code cloning, and static path feasibility analysis in combination with dynamic analysis to identify correct (non-obfuscation) control flow edges in the disassembly. Christodorescu and Jha (2003) demonstrated that simple obfuscation techniques could be employed to subvert basic anti-virus methods, and presented an architecture based on identifying malicious patterns that was resilient to these obfuscations. Kruegel et al. (2004) uses a technique that uses each byte of the program as a possible starting point of disassembly, then eliminates those disassemblies that are impossible or unlikely to be valid to defeat the work proposed by Linn and Debray (2003).

2.2 Packing

Many modern malware are transmitted in "scrambled" form—either encrypted or packed—in an attempt to evade detection. The scrambled code is then restored to the original unscrambled form during execution. Here, *encryption* refers to the use of some kind of invertible operation, together with an encryption key, to conceal the malware. *Packing* refers to the use of compression techniques to reduce the size of the malware payload, which has the side effect of disguising the actual instruction sequence. The distinction between these approaches is not typically important for our purposes. Hence, both will be referred to generically as "packing." The code used to transform the binary to its scrambled form is referred to as a *packer*, and the code that undoes the scrambling is called the *unpacker*.

The use of packers poses a problem for security researchers, because in order to understand how a new virus or worm (or a new variant of an old one) works, it is necessary to reverse engineer its code. If the code is packed, then it must be unpacked as part of this reverse engineering process. In some cases, it may be possible to use syntactic clues to identify the packer used on a program as in PEid (2008). If a known unpacker exists, such as for many commercial packing tools like UPX (see Oberhumer et al. (2008)) and ASPack (see ASPack Software (2009)), it can be used to unpack the file. However, malware writers can close this obvious hole by deliberately altering the "signatures" of the packer in the packed binary, or by using their own custom encryption and decryption routines.

When confronted with a malware binary packed with an unknown packer, therefore, researchers rely almost exclusively on dynamic analysis techniques to identify its code, e.g., by running the malware binary under the control of a debugger or emulator (see Szor (2005); Rogue Warrior (1996) for general description of such techniques). The Renovo project (see Kang et al. (2007)) keeps track of the state of the memory map during execution. Memory locations are considered "clean" until written to, at which point they are marked "dirty." If a dirty memory location is executed, this is treated as an indication of unpacking, and the state of memory at

this point is captured. The dirty location can then be used to begin a recursive disassembly of the unpacked code. After this analysis, the important information can be saved, and the memory marked clean, again, so that multiple phases of unpacking can be handled in succession.

OmniUnpack (see Martignoni et al. (2007)) uses a similar dynamic approach that tracks written as well as written-then-executed memory pages. However, OmniUnpack integrates its analysis into an existing malware detection engine. Its approach is to wait for potentially damaging system calls, then call the malware detection engine on the written memory pages. Only if the malware detector returns a negative result does it allow execution to continue. This approach allows for much faster run-times, and is robust to different packing algorithms.

Unfortunately, these dynamic techniques have a number of shortcomings. Execution of malware code may allow the malware to "escape." In some cases, elaborate infrastructure is required to prevent this, e.g., dynamic analysis of bluetooth-enabled devices carried out in a giant Faraday cage to prevent accidental infection through wireless transmission (see Hypponen (2007)). Dynamic techniques can also be tedious and time-consuming, and new malware can spread very quickly (see Mahadik). The more time it takes to analyze these new threats, the longer the threat can spread unabated. Finally, and most importantly, dynamic analyses are vulnerable to conditional execution of the unpacker routine. Malware may deploy anti-debugging and anti-monitoring defenses (see Danielescu (2008); Black Fenix; Cesare (1999); Julus (1999)), and skip the unpacking routine if it finds its execution is being monitored, or it could be designed to execute only under certain environmental conditions, such as a particular date. The cited example of Renovo will not unpack the malicious code if the program has chosen not to unpack it. And while OmniUnpack will allow safe execution in this case, it will also not be able to identify the malicious code. In the case of security analysis where program understanding, and not just safe execution, is required, this means handling the dynamic defenses, which can be done with the application of static analysis techniques.

Figure 2.1 shows an example of a simple unpacker, in this case for the Hybris-

Instr	*Address*	*x86 assembly code*
	0x401000:	{*... encrypted malware body ...*}
...		
I_0	0x4064a8:	movl %edx ← $0x152a
I_1	0x4064ad:	movl %eax ← $0x401000
I_2	0x4064b2:	movl %esi ← $0x44b3080
I_3	0x4064b7:	subl (%eax) ← %esi
I_4	0x4064b9:	addl %esi ← $0x2431400
I_5	0x4064bf:	addl %eax ← $4
I_6	0x4064c2:	decl %edx
I_7	0x4064c3:	jne .-0xc
I_8	0x4064c5:	jmp 0x401000

Figure 2.1: Unpacker code for Hybris-C worm

C email worm. Instructions I_0 and I_1 load registers with the size (5418 words) and start address (0x401000) of the region to be unpacked. Instruction I_2 loads the encryption key. I_3 through I_7 iterate over each word of the region, performing the decryption by means of the subtract instruction, and rotating the key value with addition. When the value of the `%edx` register (the number of words left to decrypt) is greater than zero, execution jumps back to I_3. At zero, it falls through and branches to the unpacked code. Other unpackers may differ from this code in various aspects, but they all share the property that they modify memory to create new code that was not present (in that form) in the original binary, then execute the code so created. This observation forms the key to our approach to static unpacking.

Unfortunately, at the level of a program binary, code bytes may be indistinguishable from data bytes. Chang and Atallah (2001) show that in some cases, the bytes at a memory address may be used for both. Hence, it may not be possible to tell which memory writes target code, and which modify data. Some researchers such as Maebe and De Bosschere (2003) have addressed this question using heuristics, e.g., by considering all writes to the text section of a binary as a code modification. This approach does not always work, since code can be generated in memory regions other than the text section, e.g., the data section or the heap, and the text section can contain embedded data whose modification does not, intuitively, con-

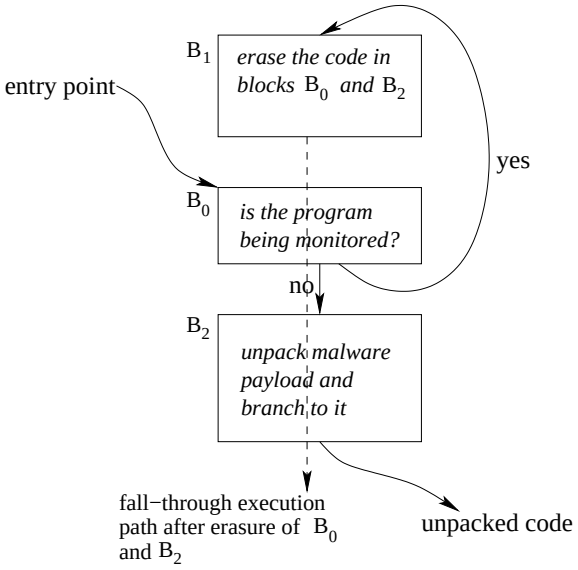


Figure 2.2: A schematic of a program that can modify its code in different ways

stitute code modification. Furthermore, packers may rename sections arbitrarily, e.g. UPX-packed binaries typically have sections named `UPX0`, `UPX1`, etc. It may not even be possible to tell, by simple inspection, which sections contain executable code. ASPACK (see ASPack Software (2009)) changes the flags in the section header table of the packed binary so that no section appears to be executable. For these reasons, we consider a more behavioral approach, where a location is considered to be code if it is possible for execution to reach that location. As we will see, however, even this is difficult to identify statically, and we will have to resort to conservative approximations.

In general, a program may modify its code in different ways and enter the modified code from different program points. Figure 2.2 shows a simple example. Execution begins at block B_0 and immediately checks to see whether it is being monitored. If it is not, control falls through to the unpacker code in block B_2, which unpacks the malware payload and branches to it. If monitoring is detected, however, control branches to block B_1, which overwrites the code in blocks B_0 and B_2 with no-ops—thereby presumably covering its tracks. Execution falls through the no-op sequence

into unrelated code following B_2. In this example, there are essentially two unpackers: the one in B_2 which unpacks the code and branches to it, and the one in B_1 which hides the malicious intent of the program and falls through to the unrelated code. In general, there may be an arbitrary number. Thus, our approach focuses on identifying points in the program where control can enter unpacked code, and treats the unpackers for each case separately.

Our approach, presented in Chapter 3 will be behavior based, so as to handle unknown packing techniques. It will also use a static analysis approach so as to handle dynamic defenses, and be able to identify multiple unpackers as in the example of Figure 2.2.

We use the value-set analysis described in Balakrishnan (2007) and Balakrishnan and Reps (2004) in our approach. There is a great deal of research literature on similar alias analysis, for example, the discussion by Hind and Pioli (2000) and the survey by Rayside (2005). The analysis that we use is specifically designed for use in unstructured assembly language code. Additionally, we use backward static slicing to identify parts of the code critical to the process of unpacking. There is also considerable information in the literature about slicing techniques, for example, see the survey papers by Tip (1995) and Xu et al. (2005). Bergeron et al. (1999) discuss similar work in the use of static slicing techniques for identifying malicious behavior in unpacked binaries. However, we are not aware of the application of any of this work towards automatic unpacking of malware.

2.3 Virtualization-Obfuscation

Recent years have also seen an increase in malware protected against analysis and reverse engineering using virtualization obfuscators such as VMProtect (see VMProtect Software (2008)) and Code Virtualizer (see Oreans Technologies (2008)). Such obfuscators embed the original program's logic within the byte code for a (custom) virtual machine (VM) interpreter. The VM then fetches and interprets its virtual instructions. The example shown in Figure 2.3(a) shows the control flow

graph (CFG) of a simple 6 instruction program that contains one loop. Figure 2.3(b) shows what that program may look like after being transformed by a simple, but typical, virtualization obfuscation. Each of the original instructions has been transformed into a virtual instruction for the virtual machine. The virtual machine fetches the byte code of the next instruction to execute, then uses a table to look up the address of the virtual instruction implementation corresponding to that byte code. The effect is that the control flow of the loop from the original program is completely hidden to static analysis. The control flow of the transformed program is that of the virtual machine's dispatch routine, and the original control flow is embedded in the byte code.

We wish to point out that this notion of obfuscation through virtualization is very similar to the idea of *control flow flattening* by Wang et al. (2001). In control flow flattening, the control flow of the program is altered in a two-step process. First, high-level control structures are transformed into if-else-goto constructs. Next, the goto statements are replaced with switch statements, where the switch variable is dynamically set within the if-else structure. The effect on the code is that any basic block in the control flow graph may be a predecessor or successor to any other basic block in the graph. The resulting analysis of the control flow then is transformed into a data flow problem. That is, identifying the value of the switch variable at each entry into the switch statement.

From a more theoretical, and more general, point of view, a virtualization obfuscator takes a program P and produces a pair $V_P = (B_P, I_P)$ where B_P is an interpreted representation of P and I_P is an interpreter for a custom virtual machine whose sole purpose is to execute the program B_P. The virtual machine implemented by I_P typically has very little resemblance to the underlying hardware. A common choice of representation is byte code, in which case the interpreter I_P is byte-code interpreter that uses the familiar fetch-decode-execute loop: it repeatedly fetches an instruction opcode, decodes it, and dispatches execution to the code fragment that handles the operation specified.

What makes this kind of obfuscation especially challenging to deal with is that

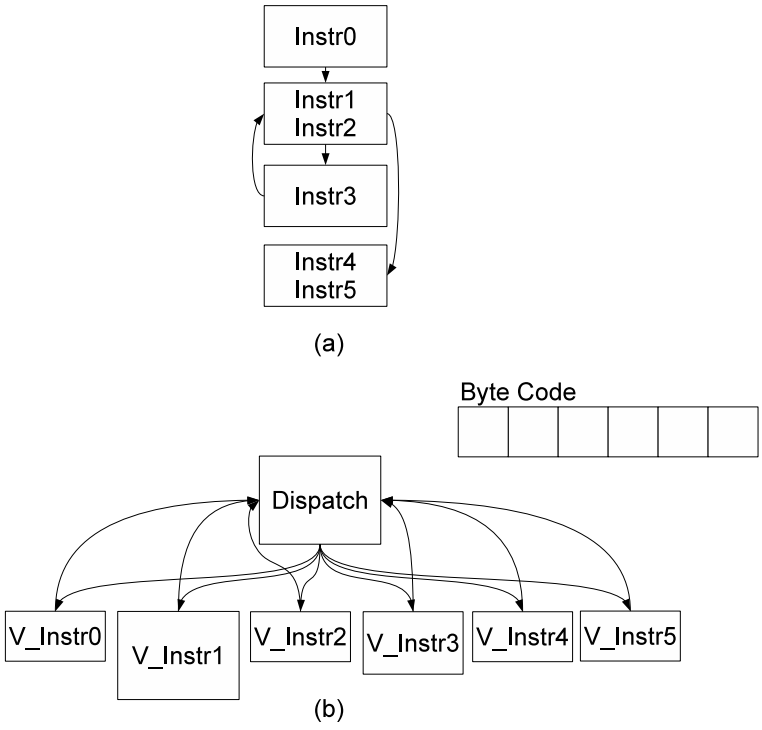


Figure 2.3: Virtualization of a simple program.

examining the code for the interpreter I_P tells us very little about the logic of the original program P, which is actually hidden in the byte code program B_P. For example, an execution trace for a the virtualized program V_P will show only the instructions in the interpreter I_P. Memory accesses in this trace will be a mixture of accesses reflecting the data manipulation behavior of the original program P and those pertaining only to the operation of the interpreter I_P—it will be difficult to tease these apart. Control transfers in the trace, similarly, will be a mixture of those stemming from the logic of P and those corresponding to the dispatch loop of I_P.

Existing techniques for reverse engineering virtualization-obfuscated code first reverse engineer the VM interpreter; use this information to work out individual byte code instructions; and finally, recover the logic embedded in the byte code program. Falliere (2009) apply this "outside-in" approach specifically to the Clampi virus. Sharif et al. (2009) use a general version of this approach that is applicable to many virtualization-obfuscation protected binaries. They identify the virtual program counter, and in turn identify what locations in memory contain the implementation of the original instructions. They can then read the memory and rebuild an approximation of the original program. Similarly, Rolles (2009) dynamically identify the entry points into each virtual machine, then must reverse engineer the VM by building a disassembler for the byte code.

This outside-in approach is very effective when the structure of the interpreter meets certain requirements. However, when the interpreter uses techniques that do not fit these assumptions (e.g., direct-threading vs. byte-code interpretation), the deobfuscator may not work well. This approach may also not generalize easily to code that uses multiple layers of interpretation. The reason for this is that a fundamental component of this approach is the identification of specific sets of memory accesses in an execution trace as being byte-code fetches, and distinguishing between instruction fetches for the various different interpreters in the different layers seems challenging.

This dissertation presents a different, behavioral approach to the problem that uses dynamic analysis techniques. First, Chapter 4 discusses how the behavior of

the original program can be identified and captured for a particular instance of execution using a dynamic trace. Next, Chapter 5 discusses how to identify the conditional control flow of the original program so that the previous results can be generalized for multiple instances of execution (i.e., multiple input values).

Similar to this work, there has been some study in the programming language community on using a technique called *partial evaluation* (see Jones et al. (1993)) for code specialization, in particular for specializing away interpretive code. However, the literature assumes that the program analysis and transformation are static, which suggests that its application to highly obfuscated malware binaries may not be straightforward.

The work by Udupa et al. (2005) (mentioned previously) discusses techniques for deobfuscating code that has been subjected to the similar transformation known as control flow flattening. These techniques are static, and therefore very different from the ideas presented in Chapters 4 and 5.

Our work relies heavily on identifying dependencies between instructions in the dynamic trace. There is a rich body of work on various sorts of dependence analysis in the program analysis literature. For example, the notion of ud-chains for relating uses and definitions of variables during static program analysis is well-established Aho et al. (1985).

Our work also relies on an in-house implementation of a custom equational reasoning system designed for use on x86 assembly code. Equational reasoning has been applied to many problems in software analysis. Jorge et al. (2009) use it to certify properties of a functional program. Wells and Vestergaard (2000) link first class primitive modules with an equational reasoning. Gill (2006) use it to rewrite Haskel fragments.

We are unaware of any existing equational reasoning system capable of handling x86 assembly code, and are also unaware of similar applications of such a system to identifying conditional control flow in obfuscated code. However, the analysis of assembly code in general is well studied. Walenstein et al. (2006) used term rewriting for normalizing metamorphic variants of viruses and other malicious

code. This work deals specifically with identifying semantics-preserving x86 code transformations performed by malware and does not address analysis of the code in general. Leroy (2009) employed equational reasoning to verify PowerPC assembly generated by a CompCert compiler. Similar RTL-style representation has been used by Kinder and Veith (2008) to improve disassembly by identifying indirect jumps to function calls. Other examples include the efforts by Djoudi and Kloul (2008), Brauer et al. (2009), and Venkitaraman and Gupta (2004).

Finally, we note that our equational reasoning system is similar in some aspects to the formal verification work of Magnus Myreen Myreen (2009), though his work is concerned with verification of code, and ours is geared towards understanding behavior. Our approach is also similar, in principle, to symbolic execution techniques (e.g., King (1976) and Coen-Porisini et al. (2001)). However, symbolic execution typically does not address low level architectural issues such as register-level aliasing in the x86 architecture.

CHAPTER 3

STATIC ANALYSIS OF PACKED MALWARE

Section 2.2 describes the basic approach behind the obfuscation technique known as packing, and lists some of the problems associated with dynamic approaches to this problem. These problems motivate our search for an alternative to identifying the code generated when a packed malware binary is unpacked. In this context, our goal is to use static program analyses to construct detailed behavioral models for malware code, which can then be used by security researchers to understand various aspects of the behavior of a malware binary: how the code for a program may change as it executes; the control and data flow logic of the various "layers" of the binary that are created by successive unpacking operations; and the static and dynamic defenses deployed by the binary. This chapter presents work published in WCRE 2009 (see Coogan et al. (2009)) that takes a first step in this direction by describing a general and automatic approach to statically unpacking malware binaries. The main contributions of the work presented in this chapter are as follows:

1. It shows how well-understood program analyses can be used to identify whether a program may be self-modifying (which may indicate unpacking).

2. For programs that are found to be possibly self-modifying, it shows how the code modification mechanism, i.e., the code that carries out the unpacking, can be identified and used to unpack the binary without any prior knowledge about the packing algorithm used.

3. It shows how standard control and data flow analyses can be applied to this code modification mechanism to find (and possibly neutralize) dynamic defenses, time/logic bombs, etc. that activate the unpacking conditionally.

Our approach is intended to complement — and not necessarily replace — dynamic analysis techniques currently used by researchers for analyzing malware.

The remainder of this chapter is organized as follows. Section 3.1 discusses the theoretical basis we use for identifying the unpacker. Section 3.2 discusses the nature of the dynamic defenses that we must handle. Section 3.3 details our approach to the problem. And Section 3.4 discusses the results obtained by running an implementation of our system on several test files.

3.1 Identifying the Unpacker

In order to carry out unpacking statically, we first have to identify the code in the packed binary that carries out unpacking. In order to do this, we have to be able to distinguish between the part of the malware's execution when unpacking is carried out from the part where it executes the unpacked code. This section discusses the essential semantic ideas underlying the notion of "transition points," i.e., points in the code where execution transitions from unpacker code to the unpacked code newly created by the unpacker. This notion of transition points underlies our approach to identifying and extracting the unpacker code, which is then used to carry out unpacking.

For our purposes, it suffices to focus on an individual transition from ordinary (i.e., unmodified) code to modified code. The reason is straightforward. We know that the unpacker must modify code at runtime to affect the unpacking, and that this unpacking must occur before the code can be executed. Hence, the first instance of executing unpacked code must have been brought about the unpacker. To this end, given a program P, consider a trace of a single execution of P. This consists of a sequence of states $\mathbf{S} = S_0, S_1, \ldots$, where each state S_i consists of (i) the contents of memory, denoted by $Mem(S_i)$;[1] and (ii) the value of the program counter, denoted by $pc(S_i)$, specifying the location of the instruction to be executed next. At any state S_i, we can determine which memory locations (if any) have changed relative

[1] For simplicity of discussion, we consider registers to be a special part of memory.

to the previous state S_{i-1} by comparing $Mem(S_i)$ with $Mem(S_{i-1})$. This notion generalizes in a straightforward way to the set of memory locations modified over a sequence of states $\langle S_i, \ldots, S_j \rangle$, which we will denote by

$$ModLocs(\langle S_i, \ldots, S_j \rangle)$$

We can divide the execution trace $\mathbf{S}$ into two phases: an initial unpacking phase $\mathbf{S}_{unpack}$, followed by the subsequent execution of unpacked code $\mathbf{S}_{exec}$:

$$\mathbf{S} = \underbrace{S_0, \ldots, S_k}_{\mathbf{S}_{unpack}}, \underbrace{S_{k+1}, \ldots,}_{\mathbf{S}_{exec}}.$$

The boundary between these two phases is marked by the execution of a memory location that was modified earlier in the execution. Thus, $\mathbf{S}_{exec}$ begins (and $\mathbf{S}_{unpack}$ ends) at the first state S_{k+1} for which $pc(S_{k+1}) \in ModLocs(\langle S_0, \ldots, S_k \rangle)$; if no such S_{k+1} exists, no unpacking has taken place on this execution. If we assume complete knowledge about the trace $\mathbf{S}$, we can give an idealized definition of the set of unpacked locations, UL_{ideal}, as those locations in $ModLocs(\mathbf{S}_{unpack})$ that are subsequently executed:

$$UL_{ideal}(\mathbf{S}) = ModLocs(\mathbf{S}_{unpack}) \cap \{pc(S_i) \mid S_i \in \mathbf{S}_{exec}\}.$$

In practice, of course, we do not have *a priori* knowledge of the set of locations that will be executed after unpacking (in fact, until unpacking has been carried out, we do not even know which locations *could* be executed after unpacking). We therefore use the set of memory locations modified during the unpacking phase up to the point where control enters an unpacked location, i.e., the set of $ModLocs(\mathbf{S}_{unpack})$, as a conservative approximation to the idealized set of unpacked locations $UL_{ideal}(\mathbf{S})$. We then define the dynamic unpacker $\mathcal{U}_D$ for the trace $\mathbf{S}$—i.e., the code that actually carries out the memory modifications in the unpacking phase of this trace—to be the fragment of the program P that was executed during $\mathbf{S}_{unpack}$ and which could have affected the value of some location in $ModLocs(\mathbf{S}_{unpack})$. This is nothing but the dynamic slice of P for the set of locations $ModLocs(\mathbf{S}_{unpack})$ and the execution trace $\mathbf{S}_{unpack}$.

There are two key pieces of information used to define the dynamic unpacker here: the state where control is about to flow into unpacked code, and the set of memory locations that get modified by the time control reaches this state. For static analysis purposes, we consider the natural static analogues for these. Reasoning analogously to the distinction between $\mathbf{S}_{unpack}$ and $\mathbf{S}_{exec}$ above, we find a pair of locations (ℓ, ℓ') such that control can go from ℓ to ℓ', and ℓ' may have been modified earlier in the execution. We refer to such pairs as transition points:

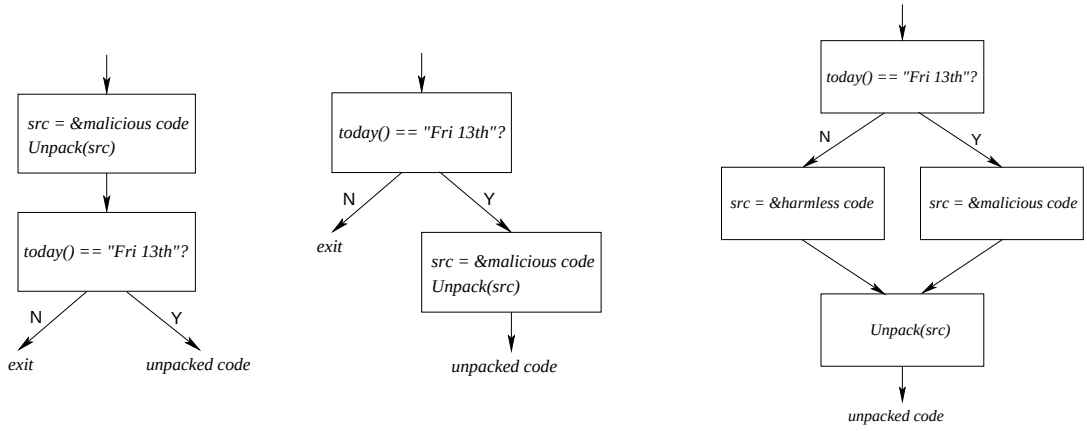
Definition 3.1.1 A *transition point* in a program P is a pair of locations (ℓ, ℓ') satisfying the following:

1. ℓ is not modified during the execution of P;
2. there is an execution path from the entry point of P to the point ℓ along which ℓ' may be modified; and
3. the next instruction to be executed after ℓ may be at ℓ'.

∎

Intuitively, a transition point (ℓ, ℓ') gives a static characterization of the point where control goes from the unpacker to the unpacked code: ℓ corresponds to the program counter in S_k, the last state in the unpacking phase $\mathbf{S}_{unpack}$ in the trace shown above, while ℓ' corresponds to the program counter in S_{k+1}, the first state of the unpacked code execution phase $\mathbf{S}_{exec}$.

Given a transition point t for a program P, let $Mods(t)$ denote (an upper approximation to) the set of memory locations that may be modified along execution paths from the entry point of the program to t. We define $\mathcal{U}_S(t)$, the static unpacker for t, to be the static backward slice of P from the program point t with respect to the set of locations $Mods(t)$, i.e., the set of instructions whose execution could possibly affect any of the locations in $Mods(t)$.



(a) simple (b) control-based (c) data-based

Figure 3.1: Different kinds of dynamic defenses

3.2 Handling Dynamic Defenses

As mentioned earlier, one of the drawbacks with dynamic analysis of malware binaries is that it allows the malware to deploy dynamic defenses. Examples of such defenses include anti-debugging code, which attempt to detect whether the program's execution is being monitored; time bombs, which cause the malware to be activated only at certain times or dates; and logic bombs, which activate the malware upon the detection of some environmental trigger.

We can classify dynamic defenses into three categories, whose conceptual structures are shown in Figure 3.1. Here, the variable *src* refers to the packed code. The first kind, shown in Figure 3.1(a), is *simple*: here, the dynamic defense predicate is executed after the malicious code has been unpacked. The second kind, shown in Figure 3.1(b), is *control-based*: here the dynamic defense predicate is executed first, and the unpacker is invoked conditionally based on the outcome of this test. The *W32.Divinorum* virus attempts to use such a technique (see Ferrie (2008)), though a bug in the code renders the defense ineffective. Finally, Figure 3.1(c) shows *data-based dynamic defense*, whose effect is to pass different values to the unpacker based on the outcome of the test. As a result, the outcome of unpacking is different based

on whether or not the dynamic defense predicate is true.

Many of the dynamic defenses currently encountered in malware use the simple defense shown in Figure 3.1(a). Existing emulation-based techniques are sufficient to identify the malware in this case, since the malicious code is materialized in unpacked form in memory regardless of whether or not it is executed. We therefore do not consider such defenses further. However, *control-based* and *data-based* may cause a dynamic analyzer to miss, or incorrectly unpack, the true malware, thus leading to tedious and time-consuming manual intervention.

Using static analysis, we can use the control-flow structure of the malware code to detect dynamic defenses. To this end, we recall the notions of dominators and post-dominators from static control-flow analysis (see Aho et al. (1985)). Given two basic blocks B and B' in the control flow graph of a program P, B *dominates* B' if every execution path from the entry point of P to B' goes through B. B *post-dominates* B' if every execution path from B' to the exit node of P passes through B. We can use these notions to identify dynamic defenses as follows:

- **Control-based defenses:**
  Unpacking is control-dependent on a conditional branch C if the unpacker code is reachable from C but does not post-dominate C.

- **Data-based defenses:**
  Unpacking is data-dependent if the instructions that define the unpacking parameters do not dominate the unpacker code.

In the first case, since the unpacker code does not post-dominate the conditional branch C, control may or may not reach the unpacker at runtime depending on the outcome of C. In the second case, since the instructions defining the unpacking parameters do not dominate the unpacker, different execution paths can assign different values for these parameters. Note that in both cases these are necessary but not sufficient conditions (the usual undecidability results for static analysis make it difficult to give nontrivial sufficient conditions).

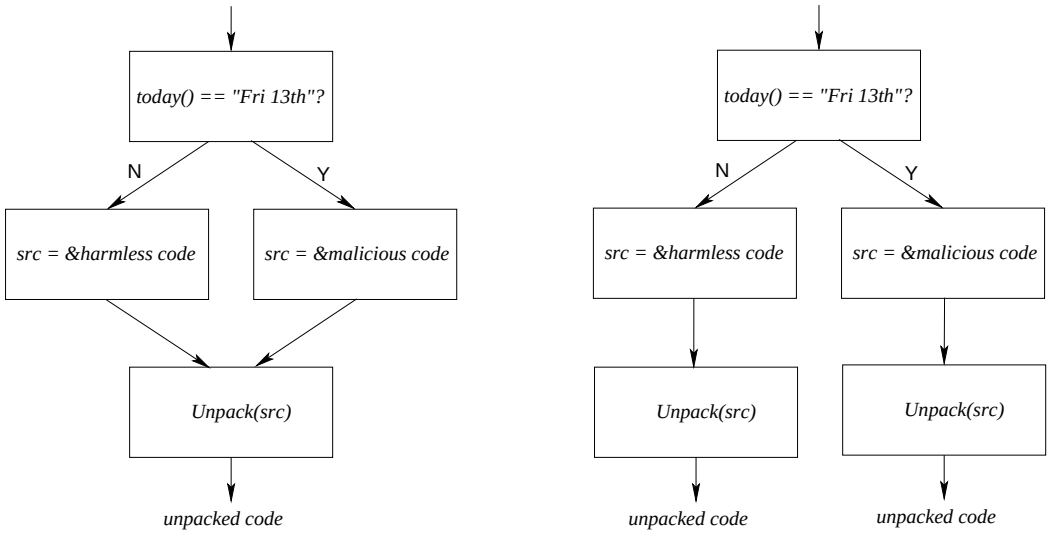


Figure 3.2: Transforming data-based dynamic defense to control-based defense.

It turns out that a data-based dynamic defense can be transformed to a control-based one using a code transformation known as *tail duplication* (see Mahlke et al. (1992)). This is illustrated in Figure 3.2. From the perspective of static unpacking, therefore, data-based dynamic defenses can be handled by transforming them to control-based defenses and then handling these as discussed in Section 3.3.2. The remainder of this chapter therefore focuses on dealing with control-based dynamic defenses.

While dynamic defenses can be detected as discussed above, in general they cannot always be completely eliminated while preserving the unpacking behavior. To see this, consider the situation where an external input is read in and used as a password and also as a decryption key: in this case, eliminating the dynamic defense would be equivalent to automatically guessing the password. However, automatic elimination of dynamic defenses may be possible if the value that is tested in the dynamic defense predicate is unrelated to the decryption key(s) used for unpacking (here, the two are considered to be "related" if there is some value v such that, for some functions f and g, the value $f(v)$ is used in the dynamic defense predicate and $g(v)$ is used as an unpacking key). This is usually true of malware code, where the

dynamic defense is related to some aspect of the external environment, e.g., execution under the control of a debugger or in a virtual machine, while the unpacking key is typically stored within the program executable itself.

3.3 Our Approach

The overall organization of our static unpacker is as follows:

1. *Disassembly and control flow analysis.* Read in the input binary and use information about the program entry point (found in the file header) to obtain an initial disassembly of the binary. We perform control flow analysis using standard techniques to identify basic blocks and construct the control flow graph of the disassembled code (see Aho et al. (1985)).

2. *Alias analysis.* Perform binary-level alias analysis to determine the possible target addresses of indirect memory operations in the disassembled code. Our current implementation uses the value-set analysis of Balakrishnan (2007), and Balakrishnan and Reps (2004), and is described in brief in Section 3.5 at the end of this Chapter.

3. *Potential Transition point identification.* Use the results of alias analysis to identify potential transition points, i.e., points where control may be transferred to unpacked code (see Definition 3.1.1). We refer to these as "potential" because imprecision in the alias analysis may identify some locations as possible transition points even though in reality they are not.

4. *Static unpacker extraction.* For each potential transition point t identified above, we use the results of alias analysis to determine the set of memory locations that may be modified along execution paths to t, and use backward static slicing on this to identify the static unpacker $\mathcal{U}_S(t)$.

5. *Static unpacker transformation.* Various analyses and transformations are applied to the unpacker $\mathcal{U}_S(t)$, extracted in the previous step, to enable it to

be executed as part of a static unpacking tool. These include the detection and elimination of dynamic defenses that effect control-dependent unpacking, as well as address translation and code change monitoring.

6. Finally, the transformed code is invoked to effect unpacking.

3.3.1 Potential Transition Point Identification

As outlined above, we begin by disassembling the binary, then carrying out alias analysis for all indirect memory references (the targets of direct references are readily apparent, and do not need additional analysis). Given the aliasing information, for each instruction I we compute two sets: $write(I)$, the set of memory locations that may be written to by I; and $next(I)$, the set of locations that control may go to after the execution of I. These sets are computed as follows, with $alias(x)$ denoting the possible aliases of a memory reference x:

$$write(I) = \begin{cases} \{a\} & \text{if } I \text{ is a direct write to a location } a; \\ alias(r) & \text{if } I \text{ is an indirect write through } r; \\ \emptyset & \text{otherwise.} \end{cases}$$

$$next(I) = \begin{cases} \{a\} & \text{if } I \text{ is a direct control transfer to a location } a; \\ alias(r) & \text{if } I \text{ is an indirect control transfer through } r; \\ \{addr(I')\} & \text{otherwise, where } I' \text{ follows } I \text{ in the instruction} \\ & \text{sequence.} \end{cases}$$

We next identify potential transition points, which indicate points where control may go from the unpacker into the unpacked code (i.e., modified locations). More formally, the idea is to collect all instructions I such that there is some instruction J that can modify some location in $next(I)$ and where there is a control flow path from J to I. Imprecision in the alias analysis will lead to multiple potential transition points. We extract and execute a slice for each one to identify true transition points.

3.3.2 Static Unpacker Extraction

Once potential transition points have been identified as described above, we process each transition point t in turn and extract the corresponding unpacker $\mathcal{U}_S(t)$. To this end, let *ep* denote the entry point of the program (i.e., instruction sequence) P under consideration, and define the set of memory locations $Mods(t)$ that may be modified along some execution path leading to t as follows:

$$Mods(t) = \bigcup\{write(I) \mid I \in P \text{ is reachable from } ep \text{ and } t \text{ is reachable from } I\}.$$

The unpacker $\mathcal{U}_S(t)$ associated with t is then computed as the backward static slice of the program from t with respect to the set of locations $Mods(t)$. Note that because of the unstructured nature of machine code, slicing algorithms devised for structured programs will not work; we use an algorithm, due to Harman and Danicic (1998), intended for unstructured programs. Since the computation of this slice considers all of the memory locations that can be modified in any execution from the entry point of the program up to the point t, most of the code in the initial disassembly is usually included; however, obfuscation code that is dead or which has no effect on any memory location will be excluded.

Once identified, we want to use the unpacker to unpack the code. However, in its raw form it is not suitable for execution. For example, virtual addresses in the code will not point to their intended locations since the unpacker will be loaded into allocated memory on the heap. Additionally, any dynamic defenses will still be included and may disrupt the unpacking process. Our next step, therefore, if to transform the code to a form suitable for execution using the following transformations.

Instruction Simplification

The Intel x86 architecture (targeted by a great deal of malware because of its ubiquity) has a number of instructions with complex semantics and/or *ad hoc* restrictions. The simplification step rewrites such instructions, which are difficult to handle

during the address translation step (see below), to an equivalent sequence of simpler instructions.

As an example, the *repz/repnz* prefixes on certain string instructions cause repeated execution of the instruction. The effect of the prefix is to decrement the `%ecx` register, then—depending on the prefix, the value of `%ecx`, and the result of the last comparison operation in the string instruction—either repeat the execution of the string instruction, or else exit the repetition. The problem here is that the side effect of the *repz/repnz* prefix on the `%ecx` register interferes with register save/restore operations in the address translation step. We address this by replacing the *repz/repnz* prefix with explicit arithmetic on the `%ecx` register together with a conditional jump that re-executes the string instruction when necessary.

Dynamic Defense Elimination

As mentioned in Section 3.2, data-based dynamic defenses can be transformed to control-based ones in a straightforward way, so it suffices to deal with control-based dynamic defenses. We do this as follows. After the slice $\mathcal{U}_S(t)$ has been constructed, we check each conditional branch that is in the slice to see whether it might be a dynamic defense test. For each such branch J, we check to see whether there is some instruction I in the slice that does not post-dominate J. If this is the case, we transform the code as follows:

1. if the slice is reachable along the *true* edge of J but not along the *false* edge, we "unconditionalize" J, i.e., replace J by a direct jump to the target of J;

2. if the slice is reachable along the *false* edge of J (i.e., along the fall-through) but not along the *true* edge, we remove J;

3. if the slice is reachable along both the *true* and *false* edges of J, J is left unchanged.

This process is repeated until there is no further change to the slice.

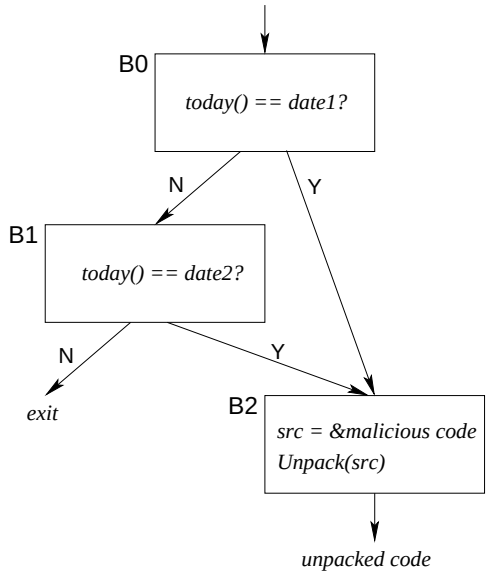


Figure 3.3: A dynamic defense with a compound predicate

The first two cases above are fairly obvious. To see the need for the third case, consider Figure 3.3, which modifies the code in Figure 3.1(b) so that the unpacker now runs on two different days. In this case, suppose that block B0 is processed first by our algorithm. The unpacker code is reachable from both its *true* and *false* branches of this test, so the test is left unchanged (case 3 above). Block B1 is processed next, and case 1 is found to apply, so the conditional branch in B1 is replaced by an unconditional jump to B2. In the next iteration, we consider block B0 again, and again find that the unpacker code in B2 is reachable along both the *true* and *false* edges out of B0, so there is no change to the slice, and the algorithm terminates. Notice that in this example, even though the test in block B0 remains as part of the slice, the dynamic defense has effectively been disabled: control goes to the unpacker code regardless of the outcome of this test.

As noted earlier in Section 3.2, it may not be possible to eliminate a dynamic defense if the value tested in the defense predicate is related to a value used for unpacking. We can identify this situation by examining data dependency relationships in the slice code.

Sandboxing

Once dynamic defenses have been eliminated, we further transform the code to ensure that memory accesses are handled correctly. There are two components to this: *address translation*, which redirects accesses to global memory regions (code and static data) to the appropriate locations; and *stack shadowing*, which deals with stack accesses from the malware code.

I. Address Translation The need for address translation arises out of the fact that the runtime unpacking of a malware binary takes place within an executing malware file, while in our static unpacking tool it occurs within a tool where each section comprising a malware binary is represented as a dynamically-allocated data object. Each such object—which we refer to as an *s-object* (for "section object")—contains meta-data about the section it represents, such as its name, size, virtual address, etc., as well as the actual byte sequence of the contents of the section (where appropriate). These section meta-data are obtained from the section header table of the binary. Because of these different representations, memory references in the unpacking code $\mathcal{U}_S(t)$—which refer to virtual addresses in the malware binary, e.g., 0x401000 for the Hybris code of Figure 2.1—have to be translated to addresses that refer to s-objects in the static unpacker's memory.

We achieve this translation by traversing the instruction sequence resulting from the instruction simplification step, discussed in the previous section. For each instruction that accesses memory (except those that access through the stack pointer), the following instrumentation and transformation is performed. First, a new instruction is added that calculates the virtual address used by the original instruction and stores the result in a register r_0. The function *VirtualAddr2UnpackerAddr*() is called with the value in r_0 as a parameter, and returns the value of the corresponding address in static unpacker memory. The return value is stored back into some register r_1 (it could be r_0, but doesn't have to be), and the original instruction is transformed so that it accesses memory indirectly through r_1. Finally, instrumentation code is added before and after these instructions that save and restore the values of

all registers as needed. Thus, the correct memory location is used by the original instruction, and the instruction acts on the current machine state. The implementation of *VirtualAddr2UnpackerAddr*() is straightforward. We note that virtual address space forms a contiguous block of memory addresses starting with the base address as given in the file header. Our static unpacker memory likewise forms a contiguous block of memory of the same size with a known start address, thus there is a one-to-one correspondence between virtual addresses and unpacker addresses. Translation, then, amounts to calculating the offset of the virtual address from the file header base address, and adding that offset to the start of the unpacker memory space. This approach also allows us to identify attempts to access memory outside of the program address space.

To deal with calls to library routines, we use a set of "wrapper" routines we have created for commonly-used library functions. At program start up, we construct our own version of the executable's Import Address Table (IAT) and build a mapping from these IAT functions and addresses to our known wrapper functions and addresses. This mapping is maintained as a global data structure. Function calls are handled as follows. If it is a direct call and the target is within the unpacker slice, it is rewritten to transfer control to the appropriate instruction within the slice. If it is an indirect call, we instrument the code with a call to a handler routine. At runtime, the call handler first tries to determine if the call target is within the slice. If it is, the handler returns the translated instruction. The returned value is substituted for the original value, and the call instruction is executed. If the target is not in the slice, the call handler assumes the call is a library call. In this case, if the tool is running under *cygwin* (a Unix-like environment within Microsoft Windows), and the target is one of the set of wrapper routines we have created, the wrapper library routine is called; otherwise we skip the call instruction.

II. Stack Shadowing There are two main reasons we must explicitly handle stack accesses. First, correct execution of the program may depend on values on, or below, the stack. For example, the *Peed-44* trojan uses an offset from the stack pointer to

reach below the stack to the Thread Execution Block (TEB) to access a value that is used to carry out the unpacking (the TEB is actually stored in higher addresses, but we say "below" because the stack grows towards lower address values.) Second, it is necessary to protect the static unpacker's runtime stack should the malware try to write garbage to it or use the stack in an unpredictable way. For example, the *Rustock.C* unpacker uses a number of *push* and *pop* instructions to obfuscate its code; at runtime, this has the effect of writing garbage onto the stack.

We handle these issues by allocating a region of memory, called the *shadow stack region*, that holds the contents of two contiguous memory areas from the malware code's execution environment: its runtime stack and its TEB. The stack area of the shadow stack region grows from high to low addresses, similar to the actual runtime stack; the address of its top is recorded in a global variable, the *shadow stack pointer*. We locate the TEB just below the stack, as is done in Windows.[2] Memory within the TEB area of this region, as well as the shadow stack pointer, are initialized with values one would expect when a Windows process begins execution. Additionally, code is added to slice $\mathcal{U}_S$ so that the runtime stack and shadow stack are switched immediately before each instruction and switched back immediately after the instruction.

3.3.3 Transition Point Detection

As mentioned above, not all potential transition points are actual transition points. We can test for actual transition points as follows. Execution of the static unpacker acts on and records changes to our own copy of the program memory M. Before execution, we create a read-only copy of memory M'. Further, we instrument the

[2]We know about the location of the TEB through personal experience reverse engineering the Trojan.Peed-44 virus, which reaches directly into the TEB using an offset from the stack pointer, and have verified this behavior on several other test cases. Windows documentation states only that the TEB is set up in user-mode address space (see Russinovich and Solomon (2005)). We suspect that the information is intentionally withheld to discourage direct access to the Windows data structures, and encourage access through the Windows API.

slice code so that before each instruction is executed, we can compare the contents of the current memory to the contents of the original memory at the address of the instruction. If these contents have changed, then the instruction has been modified, and we stop execution of the unpacker, otherwise we continue. This approach assumes that given a transition point (ℓ, ℓ') there is an instruction at both of the addresses ℓ and ℓ'. This may not be true, e.g. if the packed bytes of ℓ' did not disassemble to a legal instruction. In this case, we can add `nop` instructions as needed to potential targets.

3.3.4 Putting it all together

Once the slice code has been generated, transformed and instrumented as described above, we add wrapper code around it to save the appropriate components of program state on entry (e.g., stack and frame pointers, flags) and restore this state prior to exit. The resulting instruction sequence is then run through an assembler that traverses the list of instructions and emits machine code into a buffer allocated for this purpose. A driver routine in our static unpacker then executes a function call to the beginning of this buffer to effect unpacking; control returns from the buffer to the code that invoked it once unpacking is complete.

If a slice completes execution without finding a transition point, it returns control to the driver routine. The driver then restores the contents of malware memory M from the read-only copy M', and executes the next slice.

After the malware binary has been unpacked in this fashion, we still have to extract the resulting unpacked code. Since we know the transition point for the unpacker, i.e., the address of the unpacked code, we can do this by disassembling the code starting at this address. The resulting disassembled unpacked code can then be processed using standard control and data flow analyses.

Program	*Memory image size* $\mid P_{unpD} \mid$ (bytes)	*Bytes unpacked* N_{unp}	*Memory difference* Δ (bytes)	*% correct* $1 - \Delta/\lvert P_{unpD} \rvert$
Hybris-C	28,672	21,576	4	99.99
Mydoom.q	73,728	57,367	2,708	96.33
Peed-44	151,552	1,872	48	99.97
Rustock.C	69,632	22,145	0	100.0
tElock	1,974,272	6,140	8	99.99

Table 3.1: Experimental results: static unpacking

3.4 Experimental Results

To evaluate the efficacy of our ideas, we implemented a prototype static unpacker and tested it on five files – four viruses, and one non-malicious program packed with a common packer. (*i*) *Hybris-C* uses a single arithmetic decryption operation where the decryption key is changed via a rotation at each iteration of the unpacking routine, (*ii*) *Trojan.Peed-44* looks into the TEB to get the value at the top of the SEH chain. Since it has not loaded any exception handlers, this value will be -1 if it is running natively. It uses this value to calculate the start address of the unpacking, then iterates through addresses and performs a series of bit shifting and arithmetic on each. (*iii*) *Rustock* rootkit uses two sequential decryptor loops that operate on the same memory. Additionally, more than three quarters of its unpacker instructions are obfuscation code that perform various memory and stack operations which have no effect. (*iv*) *Mydoom* is packed with commercial packer UPX, and has a fairly elaborate algorithm consisting of nested loops that may write to memory under different conditions. (*v*) TextPad is not a malicious file, but we packed it with *tElock*, a program often used to hide malware. *tElock* uses the `aam` instruction, which ordinarily adjusts the result of multiplication between two unpacked BCD values. Here, it has the effect of an implicit decryption key. *tElock* also uses several anti-disassembly tricks such as jumping into the middle of instructions, near calls to load a value on the stack, and *int $0x20* instructions that appear to be in the control flow, but never execute. For all the files above, our approach does not require knowledge of the unpacking algorithm. It only needs to identify the correct slice.

Table 3.1 summarizes the results of our experiments. These numbers were obtained as follows:

1. We dump the program's memory image P_{orig} at the point where it begins execution. We execute the code in a debugger, setting a break point at the first unpacked instruction, and dump the program's unpacked memory image P_{unpD}. The size of this unpacked image $\mid P_{unpD} \mid$, is reported in column 1 of Table 3.1.

2. Column 2 gives the number of bytes unpacked N_{unp}, calculated as the number bytes that differ between P_{orig} and P_{unpD}.

3. We run our static unpacker on a file, and if it finds that a potential transition point is a true transition point, it dumps the memory image. We denote this P_{unpS}. The value of Δ given in column 3 is the number of bytes where P_{unpD} and P_{unpS} differ.

4. Column 4 gives the accuracy of static unpacking, expressed as the percentage of bytes where P_{unpS} and P_{unpD} agree.

For all programs in Table 3.1, we have verified that the differences between P_{unpS} and P_{unpD} are the result of differences in program metadata, specifically entries in the Import Address Table, and not the bytes that are actually being unpacked. (The IAT is a section of a file used to deal with the invocation of dynamically-linked library routines.) *Rustock* loads no functions from external .dll files, *Hybris-C* loads one function, *Peed-44* loads six functions, and *tElock* loads two.

During native execution, before the entry point is reached, the Windows operating system reads the functions that need to be imported, imports them, and adds their addresses to the IAT. Our emulation does not handle this step. The differences for these files are the result of the four byte addresses of these functions not getting loaded into the IAT by our static unpacker. The case of *Mydoom* is similar. UPX-packed binaries have two separate parts to the unpacker. First, original program bytes are uncompressed and loaded into memory. Second, the list of imported

functions is unpacked and the IAT is rebuilt manually. We have confirmed through manual analysis that the 2,708 bytes that differ for *Mydoom* all result from this second step. The results of Table 3.1 represent a single phase of unpacking.

The current implementation of the code does not handle input files with multiple phases of unpacking. That is, executables that unpack some code, execute it, then unpack more code, execute it, etc. However, we believe that handling such cases is a straightforward application of our current work. We handle a single phase of unpacking by disassembling the code, and identifying the unpacker code. It should be a simple matter to begin the process again. At the point where execution enters unpacked code, we treat this first instruction as a new entry point, and disassemble the code again. Next, we run a new pointer analysis, identify new transition points, etc. This approach should be robust against dependencies with the first phase of unpacking, since the state of memory at the beginning of the unpacked code will be saved, with those changes, as the unmodified memory map.

3.4.1 Handling Dynamic Defenses

To evaluate the detection of dynamic defenses, we constructed several variants of the *Hybris-C* program incorporating various control-based dynamic defenses. We varied the structure of the code so that, for different variants, the dynamic defense code appeared above, or below, or intermixed with, the actual unpacker code. In each case, the static unpacker was able to successfully identify and eliminate the dynamic defense code. In all cases, the accuracy was identical to that given for *Hybris-C* in Columns 4 and 5 of Table 3.1.

3.5 Value-Set Analysis

The value-set analysis we use in our automatic static unpacking is entirely from the work of Balakrishnan (2007) and Balakrishnan and Reps (2004). The implementation and integration of this work into our tool was performed by our collaborators Tasneem Kaochar and Gregg Townsend. We present a brief description of this work

for the sake of completeness.

The problem of pointer analysis in the context of assembly level code is similar to source code analysis, but presents its own unique issues. Perhaps most important for our analysis is resolving indirect jumps and other memory accesses. Our analysis requires that we build control-flow graph information for the unpacker routine. We use this information to identify the potential transition points in the code. Thus, the more accurate the information is, the fewer potential transition points we will identify, and the less runtime is required to analyze the code. However, if we try to be too accurate, and miss correct information, we will potentially miss the real transition point, and our analysis will be useless.

The work of Balakrishnan provides multiple features, but most important to our work is the safe calculation of the sets of possible memory locations accessed by indirection in assembly level code. This work guarantees that our analysis is correct, and is accurate enough to provide reasonable runtime results on the test files we examine. This work was originally developed for use with the CodeSurfer/x86 tool (see Balakrishnan et al. (2005)). Translating this work to our own project required altering the data structures, algorithms, and memory joining operations of the original to work with our own data structures.

CHAPTER 4

DEPENDENCE ANALYSIS OF VIRTUALIZED CODE

Section 2.3 describes the basic approach behind virtualization-obfuscation, and the issues associated with analyzing virtualized code. Static analysis reveals only the structure of the virtual machine interpreter. The original control flow information, which is now captured in the virtual machine byte-code, is not visible. In dynamic analysis, the dynamic trace of a virtualization-obfuscated executable is a mix of virtual machine interpreter instructions and instructions performing the work of the original program. It is often difficult to see the boundaries between these two sets of instructions when looking at the trace. This task becomes even harder in the case that multiple interpreters are used, or when the interpreter dispatch routine performs multiple operations (e.g., decrypting the address of the next instruction).

Current state of the art techniques use assumptions about the structure of the interpreter to identify where virtual machine instruction implementations are stored in memory, then rebuild the source code from these implementations. This approach may not perform well in the case of multiple or nested interpreters, or when other techniques for emulation are used.

Our approach, which will appear in CCS 2011 (see Coogan et al. (2011)), uses behavioral characteristics of the original program to identify instructions that are known to be part of the original code. Specifically, we note that any observable behavior of a program is the result of its interaction with the environment. In the case of Windows executable programs, this interaction is accomplished through the Windows system calls. We also note that the behavior of a particular instance of execution can be precisely defined by a trace of the system calls made, along with their argument vectors.

This chapter will detail our work at identifying the instructions that fully define the behavior of a given dynamic trace. Chapter 5 will discuss our work on provid-

ing additional information in an attempt to generalize the behavior of the original program to more than just one set of inputs.

4.1 Overall Approach

The analysis of an executable consists of the following steps:

1. Use a tracing tool such as qemu (see Bellard (2005)), OllyDbg, Ether (see Dinaburg et al. (2008)), etc. to obtain a low level execution trace that provides, at each execution step, the address of the instruction executed, details about this instruction (byte sequence, mnemonic, operands, etc.), and the values of the machine registers.

2. Identify system calls and their arguments in this trace, using a database that gives information about arguments and return values of system calls.[1] In general, not all system calls may be of interest (e.g., those occurring in program start up or exit code may not be interesting), so we allow the user to optionally indicate which system calls to consider.

3. Use the available information to carry out an analysis on the instruction trace. This analysis will flag system call instructions, as well as any instructions that affect the values of system call arguments. We refer to these instructions as *relevant instructions*.

4. Build a subtrace from those instructions that have been marked as relevant. This *relevant subtrace* approximates a dynamic trace of the original, unobfuscated code.

[1] Our current implementation uses DLL calls as a proxy for system calls, primarily because the Microsoft Windows API for DLLs is better documented and also more consistent across different versions of the Windows operating system. This generally causes the analysis to be sound but possibly conservative since not all DLL calls lead to system calls. It is straightforward to modify this to handle code that traps directly into the kernel without going through a DLL: it simply requires examining the argument values of instructions that trap into the operating system kernel, e.g., `sysenter`, to determine the syscall number and hence the system call itself.

4.2 Value-based Dependence Analysis

To motivate our approach to deobfuscation, we begin by considering the semantic intuition behind any deobfuscation process. Obviously, when we simplify an obfuscated program, we cannot hope to recover the code for the original program, for two reasons. First, in the case of malware we usually do not have access to the source code. Second, even where source code is available, the program may change during compilation, e.g., via compiler transformations such as inlining or loop unrolling, so that the code for the final executable may be different from (though equivalent to) that of the original program. All we can require, then, is that the process of deobfuscation must be semantics-preserving: i.e., that the code resulting from deobfuscation be semantically equivalent to the original program.

In the context of malware analysis, a reasonable notion of semantic equivalence seems to be that of *observational equivalence*, where two programs are considered equivalent if they behave—i.e., interact with their execution environment—in the same way. Since a program's runtime interactions with the external environment are carried out through system calls, this means that two programs are observationally equivalent if they execute identical sequences of system calls (together with the argument vectors to these calls).

This notion of program equivalence suggests a simple approach to deobfuscation: identify all instructions that directly or indirectly affect the values of the arguments to system calls; these instructions are "semantically relevant." Any remaining instructions, which are by definition semantically irrelevant, may be discarded. The crucial question then becomes that of identifying instructions that affect the values of system call arguments.

The goal of dependence analysis is to work back from system call arguments to identify all instructions that directly or indirectly affect the values of those arguments. Figure 4.1 illustrates a simple example: suppose that the last instruction, '`print x`', is a system call. Then this instruction, and those instructions before it that contribute either directly or transitively to the value of the operand `x` of this system

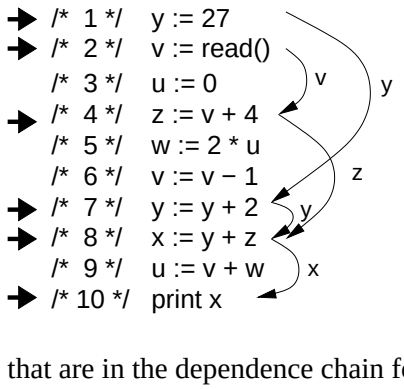


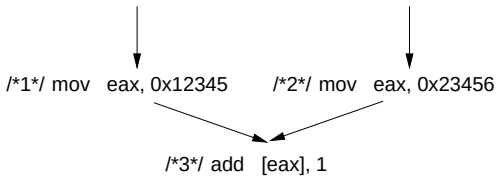
Figure 4.1: A simple example of dependence analysis.

call, are identified together with the flow of values from these instructions to the system call argument.

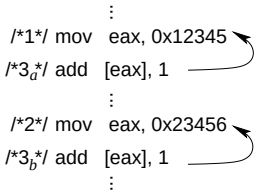
At first glance, this seems to be a straightforward application of dynamic program slicing (see Tip (1995)), but this turns out to not be the case. The problem is that slicing algorithms follow all control and data dependencies in the code (an instruction I is *control-dependent* on an instruction J if the execution of J can affect whether or not control goes to I). Since the instructions that implement a byte-code operation are all control-dependent on the dispatch code in the interpreter, it follows that the code that evaluates system call arguments and makes the system calls will also be control dependent on the interpreter's dispatch code. The net result is that slicing algorithms end up including most or all of the interpreter code in the computed slice and so achieves little in the way of deobfuscation.

We use a different approach where we initially follow only data dependencies, then consider control transfers separately. We use a variation on the notion of *use-definition* (ud) chains (see Aho et al. (1985)). Conventional ud-chains link instructions that use a variable (register, memory location) to the instruction(s) that define it. While ud-chains are usually considered in the context of static analysis of programs, it is straightforward to adapt them to dynamic execution traces.

In this case, we consider uses and definitions for each instance of an instruction in the trace separately. This is illustrated in Figure 4.2. Figure 4.2(a) shows the



(a) Static control flow structure



(b) ud relationships in the execution trace

Figure 4.2: An example of ud-chains in an execution trace

static control flow structure of a small snippet of code; corresponding fragments of the dynamic execution trace, together with ud-chains, are shown in Figure 4.2(b). Note that instruction 3 in Figure 4.2(a) occurs twice in the execution trace shown in Figure 4.2(b), denoted 3_a and 3_b respectively; each of these instances is handled separately when computing ud-chains, and so the use of eax at instruction instance 3_b does not depend on the definition of eax by instruction 1.

Because they do not follow control dependencies, ud-chains avoid the imprecision problem encountered with program slicing (control flow has to be identified separately in the deobfuscated code.) However, conventional ud-chains have precision problems of their own. The issue arises in indirect memory accesses. Consider the following instruction sequence, and suppose that registers ecx and edx have the values 0x404000 and 0x1000:

/*I_1*/	mov eax, [ecx+edx]
/*I_2*/	push eax
/*I_3*/	call `print`

Suppose that the routine print is a system call that takes a single argument. We

work back from instruction I_3, which makes this system call, find that its argument is obtained by pushing the value of register eax (I_2), then work back to I_1, where the value of eax is defined. Instruction I_1 adds the registers ecx and edx mentioned in the source operand to compute a memory address (0x405000), then loads the contents of this location into the destination register eax. The conventional algorithm for ud-chain computation would consider this instruction to "use" three different locations: registers ecx and edx and memory location 0x405000. This set is larger than it should be. The fact that registers ecx and edx are mentioned in the source operand of instruction I_1 is a happenstance of addressing that has nothing to do with the *value*–loaded from memory location 0x405000–that is passed to the system call. This results in a loss of precision. What we should do, instead, is disregard the registers used for the address computation and trace back to find the (most recent) instruction that wrote to the memory location being accessed, i.e., 0x405000.

To deal with this issue, we define a notion of *value-based dependence*. The essential intuition here is that we focus on the flow of *values* rather than on details of the intermediate computations of the addresses of these values.

This is done by redefining the set of locations used by an operand as follows:

$\mathsf{use}(op) =$
  if op is a register r **then** $\{r\}$
  else if op specifies a memory address a **then** $\{a\}$
  else $\emptyset$;

We then identify the instructions that are *relevant* to the system calls executed by the program as follows. For each system call in the execution trace, we use application binary interface (ABI) information to identify the arguments that are being passed.[2] We initialize a set S to the locations holding these arguments. We then scan back in the trace, starting at the system call, and process each instruction

[2]The ABI defines the interface between the application and the operating system. We use ABI information, along with state information from the trace, to identify what system call is being made, as well as the number, location, and value of its parameters.

I as follows: if I defines a location $\ell \in \mathsf{S}$ (which may be a register or a memory location) then I is marked as *relevant*, ℓ is removed from S, and the set of locations used by I according to our notion of value-based dependencies (see $\mathsf{use}()$ above) is added to S. This backward scan continues until S becomes empty or we reach the beginning of the trace. The effect of the value-based dependence analysis described above is that when an instruction I accesses a value from a memory location a, the dependence analysis works back to find the nearest previous instruction that wrote to location a but ignores the details of how the address a was computed by I.

4.2.1 Handling Pointer Parameters

Under certain conditions, the above algorithm may suffer from a lack of precision. The problem arises when the parameter in question is a pointer to a structure of some sort, and the function call is using an element of that structure. The trace-back based on the pointer itself only reveals the initialization of the structure. Without knowing the size of the structure, we will not recognize when elements of the structure are being set. To solve this problem, prior to performing our analysis, we must analyze the trace of the system calls to identify what values are used, and if those values are referenced using the pointer parameter. This section describes the work contributed by Gen Lu in designing and implementing a solution to this problem.

For each system call, we create a set P, which holds all of the locations (register or memory locations) which might potentially be pointers, and a set M, which holds all of the memory locations that have been accessed through a pointer. Initially, P holds the stack locations of the parameters to the call, and M is empty since we have not encountered any uses yet. We then scan forward through the trace of the system call and look at each instruction I. Typically, I will use some number of locations (i.e., register and memory locations), which we will call ℓ_1 , ℓ_2 , ..., to define some location, which we will call ℓ_d. If I uses some location $\ell_i \in \mathsf{P}$ to define ℓ_d, then ℓ_d may potentially also be a pointer and is added to P. Furthermore, if ℓ_i is known to access a memory location (e.g., eax in the instruction "$\mathsf{move\ ebx,}$

[eax]), then the value v stored at ℓ_i is also added to set M, since we know that it is a memory location accessed through a suspected pointer. Finally, if instruction I defines a location $\ell_d \in$ P, and I does not use any values from P, then we can assume that I is redefining ℓ_d as something other than a pointer that we are tracking, and we remove ℓ_d from P. The algorithm continues until P is empty, or until the end of the system call trace is reached. At this point, the set M contains a set of memory locations that we suspect are part of structures pointed to by one of the system call parameters. These locations are added to the set S above as part of the parameters passed to the call whose dependencies need to be found.

4.3 Experimental Methodology

The evaluation of our approach to deobfuscation presents several significant problems that must be addressed. In essence, these problems point back to our previous discussion of program equivalence (see Section 4.2). We have argued that *observational equivalence* is a reasonable goal, but testing for such an equivalence can be difficult. It is necessary to identify the system calls, *and* the instructions that affect their parameters. To see why the system calls alone, or the calls and the values of their parameters are not enough, consider the following example. A program that takes 2 integers and outputs their sum will produce the same output as a program that takes two integers and outputs their product, if the inputs to both programs are 2 and 2. In its simplest form, the only system call required is the print statement.

Even if we take into account the relevant instructions, we need to account for them properly. Previous work by Sharif et al. (2009) has built control flow graphs for the original program and the deobfuscated program to demonstrate similarity between the two. This approach becomes more difficult as the programs get larger and more complex. Furthermore, the idea is less applicable to our work than theirs. They use knowledge of the interpreter to identify where original instructions are stored in memory. Hence, in those cases where their code is applicable, they are able to recover most or all of the original instructions. Since we identify relevant

instructions, control flow graphs of our results will not show the structure resulting from things like dead code, or branches not taken.

To further complicate this idea, there is no guarantee that the obfuscator will use the same instructions from the original program. For example, VMProtect and CodeVirtualizer rewrite call to other semantically equivalent instructions. It is possible that obfuscators may rewrite other instructions. For example, the obfuscator may unroll some loops to hide that part of the control flow graph, or it may rewrite a multiply operation as a loop of adds, so that new control flow structure is found in the obfuscated code that is not in the original. If we consider how to quantify these differences, the problem becomes impractical.

Unfortunately, we do not have a perfect solution to the problem, so we present an imperfect solution that we try to tune to what we know about the current state of virtualization-obfuscated code. Our approach is to treat the traces and relevant subtraces as sequences. We can then use known sequence matching algorithms to compare one trace to another. This approach is robust to the idea that we cannot recover the original code precisely. Matching will give us a score for our deobfuscation, regardless of how good our results are. These scores can be compared on a relative basis. While still imprecise, a score that is significantly higher than another should correspond to better matching.

This approach is also fairly flexible, and allows us to handle several of the issues presented by program equivalence. First of all, we know that the current virtualization programs that we examined rewrite library calls using semantically equivalent instructions. It is a simple matter to replace library call implementations with a call statement at the appropriate place in the trace. Since the original code is compiled by a commercial compiler and will typically use these standard instructions, this is a reasonable step, and provides good results. This approach also allows us to handle other instances of semantically equivalent instructions. For example, it is possible that an increment instruction could be rewritten as an add instruction. We can build equivalence classes into our matching algorithm as appropriate, so that an increment matches an instruction that adds one. In doing so, we are moving

closer to the idea of comparing the behavior of two traces, and not their actual implementation. This idea is more robust and matches the intent behind program equivalence, since these cases truly are equivalent.

In addition to considering instruction operation equivalences, we must also consider how instruction operands are handled. This issue is especially relevant in the context of virtualization-obfuscated code. Due to the nature of the stack based approach used in the obfuscation programs we examined, it is possible, even likely, that the operands of the instructions will be different than in the original program. For example, in the sample files that we tested, VMProtect uses the esi register as the virtual machine instruction pointer. In CodeVirtualizer, the addresses of virtual instructions are always loaded into the al register. In both cases, the values to be operated on are stored on the virtual stack, and popped into machine registers when needed. There is no technical reason why the virtual machine would try to move these operands into the same registers that were used in the original code. To handle this, we cannot include the operands in the matching algorithm. Instead, we use only the opcode (add, call, etc.) to represent the instruction.

Next, we must consider to what we will match our results. We need to generate a trace of the original program on the same inputs. In order to present an unbiased representation of the original program, we must limit the amount of processing and analysis that is done to this trace. At the same time, we do not want to include instructions that may taint our results. As a result, we eliminate all instructions that result from library calls from both the original trace and the obfuscated trace. There are also a number of instructions that are part of the operating system initialization, and are included in every execution trace. We eliminate these instructions from both the original and obfuscated traces.

The matching algorithm itself is straightforward. Like our analysis, we use the knowledge of system calls as a guide. The traces are broken into segments, where a segment includes all instructions up to and including the next system call, or the end of the trace. In the case where the system calls between traces do not match exactly, we use the subset of calls that form a one-to-one correspondence between the two

traces. Segments are then matched, and all segments are aggregated. A matching provides a score representing how many instructions from the original trace appear in either the obfuscated subtrace or our relevant subtrace.

As a final step, we wish to calculate how effective our analysis has been. To do this, we must take into account two competing factors. First, our analysis is trying to identify as many instructions from the original trace as possible. At the same time, we are trying to eliminate as many virtual machine instructions as we can. To this end, we present two numbers for each test. The first, which we will call the *relevance score*, is the percentage of the instructions from the original trace that are identified in the relevant subtrace. The second, which we will call the *obfuscation score*, is the percentage of instructions added by the obfuscator that have been correctly excluded from the relevant subtrace. It is easy to optimize either of these values individually, but achieving good (i.e., close to 100%) scores for both is difficult, and will provide a fair evaluation of our work.

Taking into account the above discussion and concerns, we present the following methodology for evaluating our analysis:

1. Original source code of a test program is compiled into an executable.
2. A dynamic trace is generated for the original executable on some input set.
3. An original subtrace is generated by including only instructions from the executable module.
4. The executable file is protected using an available virtualization-obfuscation technique.
5. A dynamic trace is generated for the obfuscated version of the executable.
6. We perform our analysis per Section 4.1 on the obfuscated subtrace, and generate a relevant subtrace.
7. The obfuscated subtrace is matched to the original subtrace, and a score is produced.

8. The relevant subtrace is matched to the original subtrace, and a score is produced.

9. The relevance score and obfuscation score are calculated.

10. The process is repeated for all combinations of virtualization-obfuscation techniques and input test files.

4.4 Experimental Results

To evaluate our analysis, we tested the following programs. First, we wanted to test small programs that could evaluate the basics of our analysis, but which are simple enough that we could verify the results by hand if needed. We chose an iterative implementation of a factorial calculation, a matrix multiplication program implemented with double nested loops, and a recursive implementation of Fibonacci for this purpose. Next, we test two samples of malicious code – BullMoose and hunatcha. These programs were chosen for two reasons. First, we wanted to evaluate our code on actual malware. Second, we needed source code for our test programs so that we can generate our original trace from unobfuscated code, and these programs were available in C source code from the VX Heavens web site (see vxh (2011)). Finally, we tested a simple benchmark utility program that performs the md5 hashing. The results of our analysis on VMProtect obfuscated code are shown in Table 4.1 and the results of our analysis on CodeVirtualizer obfuscated code are shown in Table 4.2.

Table 4.1 shows that for most of our test programs, our analysis is able to identify better than half of the original programs instructions, while in all cases eliminating over 90% of the obfuscation introduced by VMProtect. Similarly, Table 4.2 shows even better results, identifying on average about 65% of original program instructions from CodeVirtualizer obfuscated code. However, our analysis fares worse, eliminating about 75% of obfuscation instructions on average. Overall, these results are encouraging. While 50% may not seem like a very high number for identifying original program instructions, it is useful to remember that we are only

Table 4.1: Results for programs obfuscated with VMProtect

Name	Original trace size	relevant matching	Rel. Score	Obf. Score
factorial	92	49	53.3%	98.9%
matrx_mult	651	320	49.2%	99.8%
fibonacci	151	53	35.1%	99.5%
BullMoose	94	35	37.2%	95.0%
hunatcha	2226	1118	50.2%	100.0%
md5	2257	1035	72.4%	99.2%

Table 4.2: Results for programs obfuscated with CodeVirtualizer

Name	Original trace size	relevant matching	Rel Score	Obf Score
factorial	92	50	54.4%	71.7%
matrx_mult	651	424	65.1%	82.8%
fibonacci	151	92	60.9%	91.7%
BullMoose	94	64	68.1%	72.9%
hunatcha	2226	1321	59.3%	72.5%
md5	2257	2052	90.9%	54.4%

identifying those instructions that contribute to the outward behavior of the code (through its interaction with the environment). We expect that a significant number of original program instructions are performing functions like allocating memory and initializing data structures, and will not show up in our analysis even under ideal conditions. Hand analysis of our results reveal another reason for the lower scores. We have identified several cases where both obfuscators implement certain functionality differently than the original program, and where these equivalences (see Section 4.3) were not anticipated. For example, some `push` and `pop` instructions were replaced with `mov`.

We examined the results by hand, and found the reason for the lower obfuscation scores on CodeVirtualizer files as compared to VMProtect files. CodeVirtualizer uses an interesting technique that artificially creates a dependency between some original program instructions and the virtual machine interpreter instructions. We believe that this may result from the use of constant values that are stored in memory, and

manipulated to add obfuscation. For example, instructions that require the constant 1 may instead use a reference to a variable that holds the value 1. Furthermore, if this variable is modified every time it is used (e.g., it is incremented then decremented just before use), this will create an artificial dependency between the use of the value, and all previous instances of manipulation. Of course, the obfuscator cannot change the function of the original program, so we believe these dependencies can be identified. We have had some success using code simplification techniques such as constant propagation and arithmetic simplification, but work on this issue continues.

Table 4.3: Increase in trace size due to VMProtect

Name	Original trace size	Obfuscated trace size	Relevant trace size
factorial	92	15365	212
matrx_mult	651	138798	536
fibonacci	151	16438	143
BullMoose	94	6900	373
hunatcha	2226	3327	1118
md5	2257	77219	2263

Table 4.4: Increase in trace size due to CodeVirtualizer

Name	Original trace size	Obfuscated trace size	Relevant trace size
factorial	92	172249	48706
matrx_mult	651	1571686	270064
fibonacci	151	223053	18522
BullMoose	94	120982	32803
hunatcha	2226	3881611	1066159
md5	2257	5732714	2613304

The results in Table 4.3 and Table 4.4 show the extraordinary increase in the number of executed instructions for both obfuscators. Our toy Fibonacci program, for example, executes 151 instructions in the original trace. However, the VMProtect obfuscated version executes 16,438 instructions, and the CodeVirtualizer obfuscated version executes 223,053. These numbers illustrate how important the obfuscation score is. Eliminating around 90% of the more than 5 million instructions of a potentially malicious program (as in the CodeVirtualizer protected version of the md5

checksum) means that over four and a half million instructions would theoretically not need to be examined by security analysts.

The results of matching for the obfuscated subtraces are not given in the tables. The relevant score in all cases was very nearly 100%. This result was expected, since both VMProtect and CodeVirtualizer do not eliminate instructions from the original program. Instead, they are essentially adding complexity around the original instructions. The reason that the relevant score is not exactly 100% is that some code, like function calls, is implemented using different instructions than the original. The obfuscation score for the obfuscated subtraces is, by definition, 0%.

CHAPTER 5

VIRTUALIZED CONDITIONAL CONTROL FLOW

Chapter 4 discusses the use of dependence analysis to identify instructions in a dynamic trace that are relevant to the behavior of an instance of execution of the program. This behavior can be defined by the system calls, along with their argument vectors, made by the program. The relevant instructions, then, are the system calls, plus all instructions that affect the values of the arguments passed to the calls.

However, if we want to expand our understanding of the original program beyond a single instance of execution, we must be able to say something about what decisions the program is making. In other words, we must be able to identify what branches in the program are taken, what code refers to loops and how many iterations of that loop execute, etc. This means that we must be able to identify the conditional control flow of the original program.

Identifying conditional control flow from a dynamic trace is no easy matter. We saw in the previous chapter how dependence analysis could identify the instructions that contribute to the values of the system call parameters. In that case, we only needed to know which instructions were used in the calculation. In the case of conditional control flow, this is not enough. Almost any control flow statement can be used as conditional control flow, depending on how its target address is calculated. We must know what instructions are used, and we also must know how the target address calculations are done, to know if the control flow was conditionally dependent or not.

If we could represent the target address calculations as a small number of expressions, we could easily scan these expressions for conditional dependencies. We decided to represent each instruction in the dynamic trace as an equivalent set of equations. We could then use the idea of equational reasoning. That is, we substitute terms in the equation for calculating a target address with the expressions that

define those terms, and simplify the resulting expression. We iterate this process until no terms can be substituted, and no more simplifications can be performed. The resulting expression will accurately represent the calculation of the target address. Searches for existing equational reasoning systems that could handle the unique characteristics of x86 assembly code revealed nothing, so we developed our own system.

The remainder of this chapter is organized as follows. Section 5.1 describes in detail the equational reasoning system that we developed. Section 5.2 discusses the general capabilities of our equational reasoning system. Section 5.3 discusses how we use this system for our task of identifying the conditional control flow statements in a dynamic trace. Finally, Section 5.4 discusses the results of applying the system on our overall task of deobfuscation of virtualized code.

5.1 Implementation of an x86 Equational Reasoning System

While equational reasoning has been applied in various contexts to evaluation or analysis of computer programs, until now, it has not been applied to the specific problem of analyzing x86 assembly code. This chapter presents work that will appear in SCAM 2011 (see Coogan and Debray (2011)). The x86 instruction set presents two primary issues that must be addressed in order for any such system to be useful. First, x86 instructions often have implicit functionality that must be captured, and second, the ability to define parts of registers or memory locations creates dependencies that must be correctly handled. The remainder of this section is broken into two parts. Section 5.1.1 gives an overview of our system and the rules for performing the equational reasoning. Section 5.1.2 details modifications to the system using the idea of liveness that are needed to make computation of the problem practical.

5.1.1 System Overview

We wish to be able to generate a set of equations that are equivalent to an x86 assembly code program. With this set of equations, we can choose some point in the program that is of interest to us, and use equational reasoning to derive a simplified expression that represents the value of some variable at that program point. Our system breaks this work down into the following steps:

1. Translate instructions into equations.
2. Instrument list of equations to handle dependencies.
3. Generate simplified expressions for variables of interest.

Notation

x86 assembly instructions are written in the following syntax, where *opcode* denotes the instruction operation and op_i the operands:

opcode $op_1, \ldots, op_n$

Typically, $n \leq 2$, though some instructions have additional implicit source and/or destination operands. For instructions that have both source and destination operands, the first operand is typically both a source and a destination operand: for example, the instruction

```
add  eax, ebx
```

computes '`eax := eax + ebx`'. In general an operand can be a constant value, a register, or an address expression that specifies a memory location. A special case of the latter is the indirect addressing mode, which is written as '`[`e`]`', where e is an address expression, and denotes the memory location whose address is given by the expression e. For example, an operand '`[ebx+4]`' denotes the memory location whose address is obtained by adding 4 to the contents of register `ebx`. Some instructions may also have optional modifiers, e.g., prefixes, that modify how the instructions are executed: we discuss these modifiers in later section.

We assume the existence of a dynamic trace of x86 instructions that includes information about instructions as well as the values of all user registers. There are multiple available tools that suit this purpose, and we don't discuss them further here. Each instruction in the trace is uniquely identified by its position in the trace, which we refer to as its *order number*.

Registers are treated as variables, and are given the same name as their register name. Memory locations are also treated as variables with the generic name `MLOC`: a (contiguous) range of memory locations $\{a_0, \ldots, a_n\}$ is represented as $\texttt{MLOC}[a_0 \,..\, a_n]$. Immediate values are used as is. Operations are substituted with a suitable mnemonic for understanding. The value of a (register or memory) operand *op* immediately after the execution of the instruction with order number k is denoted by op_k. We use the notation $op_\perp$ to represent the value of an operand *op* where the instruction that defined its value is not known yet. We use the notation $op_{\texttt{const}}$ when the value used has no defining instruction in the code. (this can happen, e.g., for memory initialized at compile-time). The value stored at a memory location a is denoted by $\texttt{ValueAt}(a)$.

Instruction Translation

In the first step, we make a pass over the execution trace and process each instruction as follows. We use the semantic specifications for the instruction (see Intel Corp. (2011)) to create a basic set of equations that capture the effects of that instruction, with operands represented as follows: for an instruction with order number k,

1. a destination operand *dest* is represented in the equation(s) as $dest_k$;
2. a source operand *src* is represented as $src_\perp$ (this subscript will be adjusted in the next step when dependencies are traced and the instruction that defines the value of that operand becomes known or it is found no such instruction exists).

Figure 5.1(a) gives two typical x86 instructions with their order numbers in a snippet of a dynamic trace, and Figure 5.1(b) shows the result of instruction translation

on those instructions. The code was chosen for demonstration only, and its function is irrelevant to our discussion. The "pop" operation, according to the Intel documentation, first moves the "popped" value from the memory location pointed to by the stack pointer esp to the destination location. Then, the value of the stack pointer is increased according to the size of the value moved. The and operation is translated according to its definition.

```
100:  pop eax
101:  and al, 0x4
```

(a)

$$\texttt{eax}_{100} = \texttt{ValueAt}(\texttt{MLOC}[1000\,..\,1003]_{\perp})$$
$$\texttt{esp}_{100} = \texttt{esp}_{\perp} + 4$$
$$\texttt{al}_{101} = \texttt{al}_{\perp}\ \&\ 4$$

(b)

Figure 5.1: Simple instruction translation example.

In addition, we must also account for other implicit functionality or side effects of the x86 instructions. For example, some x86 instructions also implicitly set or change the value of the eflags register. We introduce the Flag operation, which takes an expression, and returns the value of the eflags register that results from evaluating that expression. We then add a new equation to our set of the form eflags = Flag(expression) to capture this implicit behavior.

(1) $\texttt{MLOC}[1000\,..\,1003]_{100} = \texttt{esp}_{\perp}$
(2) $\texttt{eax}_{100} = \texttt{ValueAt}(\texttt{MLOC}[1000\,..\,1003]_{\perp})$
(3) $\texttt{esp}_{100} = \texttt{esp}_{\perp} + 4$
(4) $\texttt{al}_{101} = \texttt{al}_{\perp}\ \&\ 4$
(5) $\texttt{eflags}_{101} = \texttt{Flag}(\texttt{al}_{\perp}\ \&\ 4)$

Figure 5.2: Final translated equations as a result of step 1.

Finally, if the instruction being translated accesses a memory location, then additional equations are added that represent how the memory address location was calculated. These memory-location equations allow our analysis to capture and

correctly handle indirection. This is a significant obstacle in analyzing x86 assembly code since indirect memory accesses are legal in nearly all x86 instructions. Figure 5.2 shows the result of these additional steps on our example code from Figure 5.1(a).

Handling Dependencies

The purpose of this step is to associate all of the source operands in the equations from step 1 with their definitions. We do this by scanning the equation list backwards, looking for where the source is defined. If the use of a variable was always preceded by an exact definition, then this step would be trivial. However, in x86 assembly, this is not always true. Consider equations (2) and (4) in Figure 5.2. Equation (2) sets the value of the `eax` register. Equation (4) uses the value of the `al` register. Without more information, it appears that there is no relation between these two instructions. However, this is not the case. The `al` register is simply the least significant 8 bits of the `eax` register. Thus, equation (2) is determining the value of `al` used in equation (4). There are several such relationships for many of the accessible registers in the x86 architecture, as is shown in Figure 5.3. Furthermore, the x86 architecture is what is known as "byte addressable." This means that any byte of accessible memory can be written to or read from by an instruction. The x86 architecture also allows for instructions that read more than one byte at a time. These rules and relationships may make naive attempts at equational reasoning incorrect.

We note that the definition of source operands falls into one of five cases. In case I, the destination of the equation that defines the source operand is an exact match for the source operand. That is, if the source operand is `eax`, our backward search will first find an equation that sets the value of `eax`. In case II, the first equation we find in our search completely defines the source operand through a different name or reference. For example, if our source operand is `ah` and our defining equation destination is `eax`. To put it another way, the source operand is a proper *subset* of the defining destination operand. In case III, the opposite is true. The source

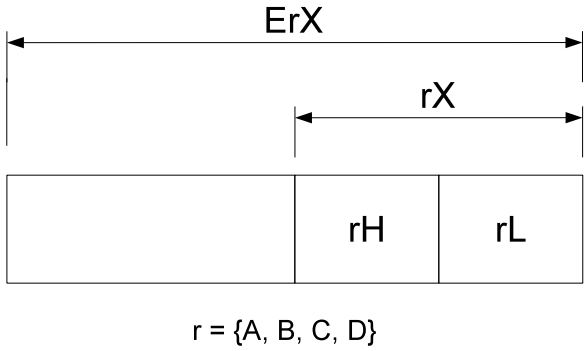


Figure 5.3: Register layout for x86 general purpose registers.

operand is a proper *superset* of the defining destination operand. This means that the full definition of the source operand comes from multiple previous instructions. Case IV occurs when there is an overlap between source operand and its definition, but neither is a subset of the other. As we will see, this case is not possible when dealing with user registers, but may occur for memory locations. Case V is where no definition is found.

In case I, the source operand is defined by a previous equation, and requires only that we note where the definition takes place. Consider the second equation in Figure 5.2. The source operand $\texttt{MLOC}[1000\,..\,1003]_{\perp}$ is passed to the `ValueAt` function. If we scan backwards in our list of equations, we see that this source is defined in the previous equation. We note the unique identifier of the defining equation, and use it to label the source operand.

In case II, the source operand is completely defined by one previous equation, but using a different name or reference. Consider the fourth equation in Figure 5.2 that changes the value of the `al` register. The source is fully defined by equation (2) in Figure 5.2. However, we are only using part of that definition. When we simplify instructions later in the process, we will want to replace this operand with its definition. However, we cannot substitute $\texttt{eax}_{100}$ because this is a four byte value, and the operand in equation (4) is a one byte value. For correctness, we must

specify which part of the value is being used. We accomplish this by introducing a new operation called `Restrict`. The `Restrict` operation takes a variable and a byte mask representing which part of the variable is to be used, and returns that portion of the value stored in the variable.

In the example of equation (4), we begin by scanning backwards through the list of equations looking for the definition of `al`. When we reach equation (2), we recognize that `al` is a subset of the definition destination. We create a new equation (marked with $*$ in Figure 5.4) after equation (2) that precisely defines the variable we are searching for in terms of how it was defined in equation (2) and using the `Restrict` function to select the part of the value that we need. We use the identifier from the defining equation to label the destination of our new equation. Figure 5.4 shows the example equations modified to handle case I and case II source definitions. The byte mask passed to `Restrict` denotes that the least significant byte (least significant 8 bits) of `eax` is used. Note that the `al` source in equation (5) changed in the same way as in equation (4). Since the same instruction calculates both `al` and `eflags`, they should appear the same in the set of equations.

(1) $\texttt{MLOC}[1000\,..\,1003]_{100}$ = $\texttt{esp}_{\perp}$
(2) $\texttt{eax}_{100}$ = ValueAt(MLOC$[1000\,..\,1003]_{100}$)
(*) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
(3) $\texttt{esp}_{100}$ = $\texttt{esp}_{\perp}$ + 4
(4) $\texttt{al}_{101}$ = $\texttt{al}_{100}$ & 4
(5) $\texttt{eflags}_{101}$ = Flag($\texttt{al}_{100}$ & 4)

Figure 5.4: Case I and case II source definitions handled correctly.

Case III occurs when multiple previous equations contribute to the full definition of a source operand. We handle this situation by adding equations to capture the first partial definition of the source operand, then continuing the search for the remainder of the definition. We have not presented an example of case III in our examples so far because it is a little more complicated, so consider the new example equations in Figure 5.5. For our discussion, we are searching for the definition of the source operands in equation (4).

(1) $\texttt{eax}_{100}$ = 1000
(2) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
...
(3) $\texttt{al}_{120}$ = $\texttt{al}_{100}$ + 4
...
(4) $\texttt{ebx}_{140}$ = $\texttt{eax}_{\perp}$ + 4

Figure 5.5: Example of case III type source dependencies.

As we search backward for the definition of $\texttt{eax}_{\perp}$ in our equations, we find it partially defined by equation (3). We cannot ignore this partial definition, of course, but neither is it a complete answer for what we need. To handle this, we add a new equation that precisely defines the source operand we are looking for in terms of the partial definition and the still unknown parts. For example, the value of `eax` at this point is the previous value of `eax`, combined with the changed value of `al`. We use simple bit-wise operators to accomplish this combination. Figure 5.6 shows this equation marked with a $*$. Note that the `eax` operand in this equation now needs a suitable definition. We recursively call our search method on this new operand until it has been fully defined. In this example, that happens when we reach equation (1), so we assign it identifier 100.

(1) $\texttt{eax}_{100}$ = 1000
(2) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
...
(3) $\texttt{al}_{120}$ = $\texttt{al}_{100}$ + 4
(*) $\texttt{eax}_{120}$ = ($\texttt{eax}_{100}$ & 1110) | $\texttt{al}_{120}$
...
(4) $\texttt{ebx}_{140}$ = $\texttt{eax}_{120}$ + 4

Figure 5.6: Case III dependencies from Figure 5.5 handled correctly.

Case IV occurs when the source operand is partially defined by a previous destination operand, and these two operands overlap but neither is a subset of the other. For the user registers, this case is impossible. If we look again at Figure 5.3, we see that in no case does a legal register name overlap another name in this way. However, in the case of memory operations, this can occur. To handle this case,

we combine our solutions from Cases II and III. First, we `Restrict` the destination operand (as in Case II) to use only that part of the destination needed by the source. Then we recursively call our function to look for the remainder of the definition (as we did in Case III).

Case V is a trivial case where no definition of a value is found. This situation can arise for memory accesses when the memory location holds a pre-initialized constant that does not change throughout the execution of the program. It can also arise for registers, especially in the case of the stack pointer. Most programs do not set the location of their stack pointer, but use the value given to them by the operating system. Thus, the initial value of the stack pointer will never be found in our algorithm. This behavior has also been seen in obfuscated code. It is possible for the programmer to use uninitialized register values as junk code to try to make program analysis more difficult. Since these values cannot be accurately predicted, they are used in such a way that they do not effect the behavior of the program. They may be ignored later, or used to generate a known value, for example by `xor`'ing the value with itself to create a value of zero. In all instances of Case V, we simply label these values at constants. That is, a source operand *src* with no explicit definition will be labeled $src_{\texttt{const}}$.

Expression Simplification

We note that at this stage, all source operands have a unique and precise definition previously in the equation list. Equational reasoning then becomes a straightforward operation. We begin by identifying what values are of interest. This could be the value of a register used in an instruction, the parameter of a function call that has been pushed onto the stack, etc. For each variable of interest, we can use an existing equation, where the variable of interest is the destination operand. If no such equation exists, we can create a trivial equation that assigns its value to itself at a point in the program where we want to analyze it (e.g., $\texttt{eax}_{103} = \texttt{eax}_{\perp}$). We then calculate the source operand definitions for the equation, as before.

To generate a simplified equation for the variable of interest, we use a simple

tree rewriting system. We begin by generating a syntax tree representation of the equation that defines our variable of interest. Each syntax tree structure has a left-hand side, represented by an operand, and a right-hand side, represented by up to two pointers to other syntax trees and one operation. The pointers point to syntax trees that represent the equations that define each of the source operands. Initially, the definitions are not known yet, so we use a simple syntax tree with only a left-hand side to represent a terminal node. If no definition is found, then this terminal node will remain. The operation is just the operation that is performed by the equation. For example, if we begin with the equation "`eax` = `eax` + `ebx`," we would generate a syntax tree with `eax` for the left hand side. The tree's operation would be labeled as "addition," and there would be pointers to two terminal node subtrees, one for `eax` and one for `ebx`.

The next step in our simple tree rewriting system is to rewrite our current tree. This is done in two steps. First, we look for a substitution to be made in our syntax tree. If we find one, then we can try to simplify the tree.

Candidates for substitution are found by traversing the current syntax tree and looking for terminal nodes. If found, we look through our list of equations for a definition of the left-hand side of our terminal node. If that is found, we generate a syntax tree representation of the new equation, and we substitute it into the original syntax tree. That is, we change the pointer that pointed to the terminal node to now point to a subtree defining that operand.

Next, we can try to simplify the syntax tree with its new information. Currently, our simplifications are fairly straightforward and simple. This is because of the nature of the code we are analyzing. Much of the code is obfuscated to make reverse engineering more difficult, and includes such approaches as hiding constant values, adding junk code, and changing control flow. The result is that the function of the original code is not changed. For this reason, we have had good success with some basic mathematical simplifications.

For example, one of the simplest techniques is to combine arithmetic constants. Imagine that user register `eax` is being used in the code as a counter variable in

a loop. For a loop that executes many times, there will be many instances of an instruction such as inc eax. After translating these instructions into equations, and substituting values, we will likely end up with an expression something like "$\texttt{eax}_{100}$ = $\texttt{eax}_{10}$ + 1 + 1 + 1 + 1 ..." Simplifying these constant additions is trivial, and can produce a simplified expression of the sort "$\texttt{eax}_{100}$ = $\texttt{eax}_{10}$ + 28."

Similarly, there are many Boolean operations used in x86 assembly code. In many cases, these operations can be simplified using the rules of Boolean algebra. For example, when performing many Boolean operations, it is often common to get an expression such as "$\texttt{eax}_{100}$ = $\texttt{ebx}_{10}$ & $\texttt{ebx}_{10}$." This, of course, would simplify to "$\texttt{eax}_{100}$ = $\texttt{ebx}_{10}$." Similarly, we can simplify "$\texttt{eax}_{100}$ = $\texttt{ebx}_{10}$ & 0x0" to "$\texttt{eax}_{100}$ = 0," and "$\texttt{eax}_{100}$ = ¬ (¬ $\texttt{ebx}_{10}$)" to "$\texttt{eax}_{100}$ = $\texttt{ebx}_{10}$."

This is not intended to be a complete list, of course. In fact, the work of simplification is nearly never ending. One can imagine that there are always more special cases of assembly code that can be identified as performing specific tasks. In fact, this work tends toward the problem of decompilation. One can imagine a set of several assembly language instructions that is used to load a value into a register, then call a system call. This instruction pattern could, in theory, be recognized as simply a call to a print routine. That is, the pattern of instructions could be "simplified" into a single call instruction. Similarly, another pattern of instructions could be "simplified" to a standard for loop, or if-then-else structure. Our current work gets very good results with just simple, straightforward simplification such as those described above. However, this is an ongoing process that is constantly being refined.

We continue this process, substituting operands with their defining expressions and simplifying, until no more changes can be made. At this point, we have a simplified expression that represents the variable of interest. We note that there are existing equation solving tools that could be made to work with our resulting equations, but we implement this simplification step ourselves.

The approach described in this section is somewhat naive, as it requires that the syntax tree be built "top-down." In reality, we use an approach that generates a

simplified equation for each equation in our equation list in order. The result is that the substitution step now substitutes a fully simplified equation for each of the new equation's source operands. This approach is described more fully in section 5.1.3, with an overall algorithm for our approach presented in Figure 5.14.

Equational Reasoning Example

To see how the whole process works, consider the sample x86 code in Figure 5.7. This code is inspired by some of the techniques that we have seen used in virtualization-obfuscation. Let's assume that we are concerned with the resulting value of `eax`after the instruction at order number 105 executes.

```
0:  xor ebx, ebx
1:  not bx
2:  mov eax, 0x7e5bd96f
3:  mov ecx, 0x81a42692
4:  and eax, ecx
5:  add ax, bx
```

Figure 5.7: Equational Reasoning example, x86 source code.

We begin, as discussed earlier, by translating all of the instructions into their equivalent set of equations. We label the destinations of the equations with their order numbers from the trace. We also insert a new equation for the value of `eax`at the point we are interested in its value. The result of the instruction translation phase is given in Figure 5.8.

The next step described above in this section is to associate all of the source operands with their definitions, while also instrumenting the code to handle dependencies caused by user register naming conventions in x86. The results of this step are shown in Figure 5.9. Notice the addition of the equations that calculate the `eflags` register. Also notice that the value of `eax`after the execution of the instruction at order number 5 falls into case III described above. Part of its definition comes from the definition of `ax`at order number 5, and the other part of its definition comes from the definition of `eax`at order number 4. Also notice that the source

ebx_0 = $ebx_\perp$ ^ $ebx_\perp$
$eflags_0$ = Flag($ebx_\perp$ ^ $ebx_\perp$)
ebx_1 = ¬ $ebx_\perp$
eax_2 = 0x7e5bd96f
ecx_3 = 0x81a42692
eax_4 = $eax_\perp$ & $ecx_\perp$
$eflags_4$ = Flag($eax_\perp$ & $ecx_\perp$)
ax_5 = $ax_\perp$ + $bx_\perp$
$eflags_5$ = Flag($ax_\perp$ + $bx_\perp$)
eax_6 = $eax_\perp$

Figure 5.8: Equational Reasoning example, after instruction translation.

operands of the addition at order number 5 both fall under case II described above, and require the "Restrict" function at their respective definitions. Finally, we note that the values of ebxused by the equation at order number 0 are whatever values happened to be in the ebxregister when the program began, and hence are labeled "const."

ebx_0 = ebx_{const} ^ ebx_{const}
$eflags_0$ = Flag(ebx_{const} ^ ebx_{const})
ebx_1 = ¬ ebx_0
bx_1 = Restrict(ebx_1, 0011)
eax_2 = 0x7e5bd96f
ecx_3 = 0x81a42692
eax_4 = eax_2 & ecx_3
ax_4 = Restrict(eax_4, 0011)
$eflags_4$ = Flag($eax_\perp$ & $ecx_\perp$)
ax_5 = ax_4 + bx_1
eax_5 = (eax_4 & 0xffff0000) | ax_5
$eflags_5$ = Flag(ax_4 + bx_1)
eax_6 = eax_5

Figure 5.9: Equational Reasoning example, after handling dependencies.

After handling the dependencies, we are now ready to reason about the value of eaxat our point of interest. Our equation for eaxat this point is given as "eax_6 = eax_5." If we substitute the right hand side of our equation for eax_5, we get "eax_6 = (eax_4 & 0xffff0000) | ax_5. We can continue this process until all possible

substitutions and simplifications are made. Figure 5.10 gives all explicit steps in the process until a final, simplified equation is reached. All substitutions are made using the equations in Figure 5.9, and all simplifications should be straightforward mathematical manipulations.

```
            eax6 = eax5
(Subst.)         = (eax4 & 0xffff0000) | ax5
(Subst.)         = ((eax2 & ecx3) & 0xffff0000) | ax5
(Subst.)         = ((0x7e5bd96f & ecx3) & 0xffff0000) | ax5
(Subst.)         = ((0x7e5bd96f & 0x81a42692) & 0xffff0000) | ax5
(Simp.)          = (0x2 & 0xffff0000) | ax5
(Simp.)          = 0x0 | ax5
(Simp.)          = ax5
(Subst.)         = ax4 + bx1
(Subst.)         = Restrict(eax4, 0011) + bx1
(Subst.)         = Restrict(eax2 & ecx3, 0011) + bx1
(Subst.)         = Restrict(0x7e5bd96f & ecx3, 0011) + bx1
(Subst.)         = Restrict(0x7e5bd96f & 0x81a42692, 0011) + bx1
(Simp.)          = Restrict(0x2, 0011) + bx1
(Simp.)          = 0x2 + bx1
(Subst.)         = 0x2 + Restrict(ebx1, 0011)
(Subst.)         = 0x2 + Restrict(¬ ebx0, 0011)
(Subst.)         = 0x2 + Restrict(¬ (ebxconst ^ ebxconst), 0011)
(Simp.)          = 0x2 + Restrict(¬ (0x0), 0011)
(Simp.)          = 0x2 + Restrict(0xffffffff, 0011)
(Simp.)          = 0x2 +0xffff
(Simp.)          = 0x1
```

Figure 5.10: Equational Reasoning example, substitution and simplification steps.

We see from the results of the simplification and substitution step that the value of eaxafter the instruction at order number 5 executes is equal to 1. This is a common type of obfuscation. Often times, compile time constants or immediate values in the code are translated in long, complicated calculations. Here, we can see how our process removes the complexity, and returns the simple constant value.

It is important to note that in the next to last step of this example, we added the values 0x2 and 0xffff. The system must know that this addition was originally generated by the 16-bit addition of the expression "$\texttt{ax}_4$ + $\texttt{bx}_1$," and hence requires

16-bit math. In 16-bit math, the value 0xffff is equivalent to -1. Hence, the result is 1. To handle this case, each operand has a size property that remains with the operand, even after an immediate value is substituted for a variable. This way, we can track the correct results of the mathematical operations during simplification.

5.1.2 Runtime Considerations

While the solutions presented to cases II and III (and by extension Case IV) above are mathematically sound, they present other problems during the equational reasoning portion of our system. Mainly, it is possible for these approaches to lead to an exponential increase in the size of the equations. Consider the example equations given in Figure 5.11. This example is similar to the one presented in Figure 5.5, except there are multiple changes to the value of `al`. Here, the ellipses indicate other instructions that are irrelevant to our analysis, except that they prevent the simplification of equations (2), (3), and (4) to avoid this problem. If we apply our approach to handling source definitions, we get the equations in Figure 5.12, because at each step we are adding a new equation with an `eax` term, then looking for the definition of that term.

(1) $\texttt{eax}_{100}$ = 1000
...
(2) $\texttt{al}_{120}$ = $\texttt{al}_{\perp}$ + 4
...
(3) $\texttt{al}_{140}$ = $\texttt{al}_{\perp}$ + 4
...
(4) $\texttt{al}_{160}$ = $\texttt{al}_{\perp}$ + 4
...
(5) $\texttt{ebx}_{180}$ = $\texttt{eax}_{\perp}$ + 4

Figure 5.11: Example leading to exponential explosion of equations.

However, many of these equations are redundant. Examining the original equations shows that we don't care about the full value of `eax` anywhere except at the beginning and the end. If we try to simplify equation (9) by substituting the term $\texttt{eax}_{160}$ with its definition, then repeat this process, we must handle many, many

eax terms that are not contributing to the value of the equation. In our example, this is not too important, because the original value of eax is set to a constant and can be simplified, but if this value were unknown, then our simplified equation would become longer and more complex. In severe cases, this added complexity can adversely effect run time to the point of being impractical.

(1) $\texttt{eax}_{100}$ = 1000
(2) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
...
(3) $\texttt{al}_{120}$ = $\texttt{al}_{100}$ + 4
(4) $\texttt{eax}_{120}$ = ($\texttt{eax}_{100}$ & 1110) | $\texttt{al}_{120}$
...
(5) $\texttt{al}_{140}$ = $\texttt{al}_{120}$ + 4
(6) $\texttt{eax}_{140}$ = ($\texttt{eax}_{120}$ & 1110) | $\texttt{al}_{140}$
...
(7) $\texttt{al}_{160}$ = $\texttt{al}_{140}$ + 4
(8) $\texttt{eax}_{160}$ = ($\texttt{eax}_{140}$ & 1110) | $\texttt{al}_{160}$
...
(9) $\texttt{ebx}_{180}$ = $\texttt{eax}_{160}$ + 4

Figure 5.12: Exponential explosion of equations.

To handle this problem, we introduce the idea of liveness into the search algorithm that we use to find the source definitions. Liveness refers to the notion of what variables are needed by the program. Simply put, a variable in a program is "live" if its value may be used later. We can use this idea to deal with unnecessary instances of a variable by tracking which parts of the variable are live. In our example, we saw that we had to create new equations that used eax each time we encountered a partial definition. This happened several times since each new equation had its own eax source operand. However, the identification of some of these partial definitions was incorrect, since our operand used the three leftmost bytes of eax, and the definitions defined only the rightmost byte. In truth, these equations do not define the relevant part of the source operand at all. To capture this idea, we assign a liveness value to all operands when we search for their definitions. In the general case, the entire variable is live, but in the case of our added equations, we use our knowledge

of what part of the variable is used to assign a more precise value.

(1) $\texttt{eax}_{100}$ = 1000
(2) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
...
(3) $\texttt{al}_{120}$ = $\texttt{al}_{100}$ + 4
...
(4) $\texttt{al}_{140}$ = $\texttt{al}_{120}$ + 4
...
(5) $\texttt{al}_{160}$ = $\texttt{al}_{140}$ + 4
(6) $\texttt{eax}_{160}$ = ($\texttt{eax}_{100}$ & 1110) | $\texttt{al}_{160}$
...
(7) $\texttt{ebx}_{180}$ = $\texttt{eax}_{160}$ + 4

Figure 5.13: Avoiding the problem of exponential explosion.

Applying this approach to our original example in Figure 5.11, when we search for the definition of `eax` in equation (5), we find the partial definition in equation (4). We add our extra equation as before, but when we recursively search for the definition, we tell the search algorithm that only the leftmost three bytes of `eax` are live. Thus, continuing to search backwards, we see equations (2) and (3), and recognize that they do not affect the value of `eax` at all because they do not define any of the live parts of the variable. Only when we reach equation (1) do we find a definition of our variable. The final set of equations given in Figure 5.13 produces better, more simplified, results.

We observe that our analysis uses a byte mask to mark which parts of a variable are live. Registers are of fixed size, and so a finite sized byte mask makes sense. In x86, most memory accesses are also of fixed size. However, in the case of string operations that carry a `rep`, `repz`, or `repnz` prefix, this is not the case. These prefixes can cause a string instruction to effectively access any sized memory block, by repeating the string operation. To allow for our fixed size byte mask, we handle each repeat of these string operations as separate instructions. Hence, all memory accesses can access a maximum of 4 bytes for a 32-bit program.

5.1.3 Algorithms and Analysis

As discussed in Section 5.1.1, our overall process breaks down into three major steps:

1. Translate instructions into equations.
2. Instrument list of equations to handle dependencies.
3. Generate simplified expressions for variables of interest.

Each of these steps can be performed in sequence, so that the overall runtime of the whole system is the addition of the runtime of each step. However, we note that each instruction is translated into a set of equations, and that all of the dependencies of these equations, by definition, are generated by instructions that came earlier in the trace. We also note that finding the simplified equation for a value at some point in the code essentially requires finding simplified equations for many previous values in the code. Further, if we were interested in more than one value, and those values depended on some common variable, then we would be repeating the work of calculating the shared variable's simplified equation.

We know that for our virtualization-obfuscation protected code, we will be interested in all of the target address calculations for control flow statements, and that these addresses often will have shared dependencies. Hence, we have developed an algorithm that passes over the dynamic trace, calculating simplified equations as we go, and using saved information from previous calculations. This algorithm is given in Figure 5.14.

In short, the algorithm performs the three major steps presented previously in this section, but does as much calculation as possible at each step. So, for each instruction, we build the set of equivalent equations that represent it. We can also instrument the equation list and identify all the source definitions for these new equations, since all dependencies will have already been calculated. This is represented in the algorithm by the for loop that calls FindDefinition for each source operand. Once this information is calculated, we can build a complete simplified equation for each of these new equations. This work is done by first calling BuildSyn-

```
Input: Dynamic Trace of Instructions : InstrList
Output: Simplified List of Equations : SimpEqnList

EqnList = ∅
DestMap = ∅
SimpEqnList = ∅
for each instr in InstrList:

    /* Translate Instructions */

    newEqns = BuildEquations(instr)
    AddToList(EqnList, newEqns)
    AddToMap(DestMap, instr, newEqns)
    for each eqn in newEqns

        /* Handle Dependencies */

        for each src in eqn
            definition = FindDefinition(EqnList, src)
            src→order = definition→order
        endfor

        /* Simplified Expressions (Tree Rewriting) */

        simpEqn = BuildSyntaxTree(eqn)
        change = true
        while change == true
                change = false
                replacement = FindReplaceableTerm(simpEqn, DestMap)
                if replacement != NULL
                   change = true
                   Simplify(simpEqn)
                endif
        endwhile
        Simplify(simpEqn)
        AddToList(SimpEqnList, simpEqn)
    endfor
endfor
return SimpEqnList
```

Figure 5.14: Algorithm for Equational Reasoning of dynamic trace.

taxTree, then executing the while loop that substitutes and simplifies the equation until finished.

To get an understanding of the runtime of this algorithm, we look at several key points in the algorithm, and discuss the work that is going on. We begin by looking at the call to BuildEquations and the subsequent calls to AddToList and AddToMap. BuildEquations works in linear time, since each instruction generates a fixed number of equations, and at this stage, no information is needed from earlier in the list. AddToList always appends information to the end of the list, and, thus, each calls takes constant time. AddToMap employs a simple hash map, and also has constant time insertion.

Next, we examine the call to FindDefinition. This call is nested inside three for loops, which indicates that it may contribute significantly to the runtime. However, closer examination reveals that this is not the case. The inner most for loop only loops through all the source operands of a given equation. This number is always constant, and is typically about 2. The next outer for loop examines each equation in the set of "newEqns." This set consists of the equations generated by a single instruction. While this number is not fixed, it can be treated as constant, and is typically around 3 or 4. In fact, the worst case that we are aware of is the "pusha" and "popa" instructions. Each generates 15 equations. That is, the total number of equations is about $O(E) = O(3 * I) = O(I)$, where I is the number of instructions in the dynamic trace. Hence, the number of total times that "FindDefinition" is called will be roughly $O(2 * E) = O(I)$. The work done by FindDefinition requires searching the list of existing equations for the definition of a source operand. In the worst case, this will mean searching all the way to the beginning of the list, and not finding a definition. Even for case III, as described above, there is a constant amount of work done each time a definition is found, and the function is recursively called starting at the last definition, so there will still be a max total work per call of $O(I)$. Hence, the total work done by FindDefinition is $O(I) * O(I) = O(I^2)$.

For our analysis, we can disregard the calls to BuildSyntaxTree. This call shares only the outer two loops with FindDefinition, and requires only constant work per

call, since each call is building the starting syntax tree for a standard equation of 2 or 3 operands.

Next, we examine the calls to FindReplaceableTerm. This is nested inside the while loop, which is inside the same outer two loops as FindDefinition. The while loop tries to replace a term in the syntax tree, and if successful, simplifies the tree, and repeats. Hence, the number of times that the loop iterates is dependent on the size of the *final* syntax tree. If the final tree is very large, then each term in the large tree is there as a result of a replacement, and simplification. It is difficult to know exactly how big the tree will be, as this depends on the nature of the code being analyzed, and not on the algorithm itself. However, we can assume that the tree has $O(T)$ terms. Hence, the call to FindReplaceableTerm will be called $O(I) * O(T) = O(IT)$ times. The work done by each call of FindReplaceableTerm is actually constant. When the equations are built by the algorithm, a map structure is built that maps the order number of the instruction with its first equation in the equation list. Since only terms that match exactly can replace another term, we only need to examine the equations for a particular order number. The algorithm looks up the location of the first equation with the matching order number in the DestMap structure. Then it examines a constant number of equation with that number.

Finally, we examine the calls to Simplify. This function is called the same number of times (at most, due to the conditional statement) as FindReplaceableTerm. However, Simplify does more than constant work. For each call, it traverses the syntax tree looking for terms to simplify. This is an overly safe approach, and is based on the idea that if the syntax tree changes, then we must examine the whole tree for simplifications. Thus, Simplify visits each term in the syntax tree, and tries to simplify it. If it finds a simplification, it does a constant amount of work, typically rewriting the term. The total work done by Simplify is the number of calls $O(IT)$ times the work per call $O(kT)$, where k is a constant. Hence, the total work done is $O(IT^2)$.

The total work of the whole algorithm, in the worst case, then is $O(I^2)+O(IT^2)$. If the number of instructions dominates the average number of terms per simplified

equation, then this will reduce to $O(I^2)$. If, on the other hand, the number of terms per instruction approaches the number of instructions in the trace, then the work will be $O(IT^2) = O(I^3)$. This is an extreme case, however. It would require that the simplified equations depend on almost all of the previous instructions, and that none of these terms can be simplified or combined in any way.

The above analysis is a worst case approach, and still does not truly reflect the actual runtime of the algorithm in practice. First, in order for the first term to approach $O(I^2)$, it is necessary that the search for the source operand definitions, on average, goes far enough back in the equations list to approach $O(I)$ work in the call to FindDefinition. This is typically not the case. In x86 assembly code, arithmetic and manipulation is typically done in the user registers, of which there are a very small number. Hence, x86 code usually takes the form of moving values into register, then operating on them very soon after, since to do otherwise would be tying up valuable resources. As a result, the actual runtime of the FindDefinition function is much closer to constant time. That is, no matter how big the dynamic trace is, the definition of any given source operand is usually found in about 10 to 20 equations before it is used. Of course, this is not true of every operand, but in practice it appears to be true on average. As a result, the first term in the runtime analysis is typically closer to $O(I)$.

Furthermore, the number of terms in each equation does not typically get very large. Again, due to the nature of x86 assembly code, values are calculated then used quickly. Most manipulation is of a simple nature that lends itself to simplification, such as multiple additions and subtractions to numbers and addresses. Hence, in practice, the T^2 in the second term of the runtime will very often reduce to a constant. That is, the typical simplified equation– regardless of its order number from the dynamic trace or the number of instructions in the trace–has less then 10 terms. As a result, the second term in the runtime analysis is often closer to $O(kI) = O(I)$.

Overall, a typical program will not require lots of work in the FindDefinition call or generate simplified expressions with large numbers of terms. Hence, it is not

unusual for the runtime to approximate $O(I) + O(I) = O(I)$. It should be noted however, that we do sometimes see cases in practice where the number of terms in each simplified equation seems to explode, resulting in the predicted $O(I^3)$ runtime. This typically occurs when the simplified equations cannot be adequately simplified. A common example is the use of values that have been decrypted. If the decryption routine does not lend itself to simplification (e.g., it combines `add`'s and `sub`'s with `xor` operations), and the value is used often later in the code, then the resulting simplified equations will grow larger and larger over the length of the trace.

5.2 Discussion and Capabilities of Equational Reasoning Tool

We begin with a small toy example to demonstrate our equational reasoning tool. We wrote a sample program that calculates the factorial of an input value, then prints the value to the screen. We captured a dynamic trace of the execution of our program with the input value 4, and used that trace as input to our system. Using our equational reasoning system to analyze the arguments passed to the print routine, the value that is printed out is determined to be the expression:

$$\texttt{ValueAt(MLOC[12ff78..12ff7b]}_{312}\texttt{)} = \texttt{((((1*(1))*(1+1))*(1+2))*(1+3))}$$

This indicates that the value passed to the print routine was the result of multiplying $1 \times 1 \times 2 \times 3 \times 4$, or 4!, which is correct. The result also indicates that there is a redundancy in our factorial implementation. It appears that our code is looping 4 times, and is calculating the initial value times 1. We checked our source code and confirmed that this is the case.

Next, to illustrate the use of our tool on more practical, real world situations, we present an example from our current area of research into deobfuscating virtualization-obfuscated code. Despite being a very simple concept, the implementation of the dispatch routine (discussed in Section 2.3) is often very complex and confusing.

Consider the example in Figure 5.15. This code is taken from a dynamic trace of a sample program that has been virtualized with the commercially available software

CodeVirtualizer (see Oreans Technologies (2008)). The example begins with the first instruction from the dispatch routine at order number 297. This instruction loads a byte from memory into the `al` register. The instructions at order numbers 305 through 312 do some calculation, then combine this calculation with the value in the `al` register. The instructions at order numbers 369 and 379 move this value onto the stack, then retrieve it later. Finally, the example ends with the actual jump to the virtual instruction. Notice that the jump occurs at order number 381, indicating that the dispatch routine consists of 85 instructions. As an analyst, we may be interested to know *how* the address of the virtual instruction is calculated. This may give us insight into what all of those instructions are doing, and why they are needed.

```
297:  lodsb
...
305:  mov bh, 0xa4
306:  xor bh, 0x25
307:  or bh, 0x7d
308:  sub bh, 0x72
309:  shr bh, 0x7
...
312:  add al, bh
...
369:  mov [esp], eax
...
379:  pop eax
...
381:  jmp dword near [edi+eax*4]
```

Figure 5.15: Sample code from CodeVirtualizer dispatch routine.

We can use our equational reasoning system to generate an expression for `eax` and `edi` at the point of the jump to the virtual instruction. First, we present an intermediate result in Figure 5.16. We have performed our analysis for the `eax` register, and suppressed any expression simplification except for simple substitution. This step is not normally performed, but it is useful for discussion. The resulting expression is too long to reproduce completely, but we have included those parts

relevant to the code in Figure 5.15. We can see that the value of `eax` starts with the value from memory location `0x4090f4`. This is the memory location implicitly accessed by the instruction at order number 297. We also see that this value is a constant, indicating that the value in this memory location has not changed since the program began execution. Next, we see that the arithmetic operations of instructions 305 through 309 are captured by the expression. We see that these operations are mainly combining immediate values, which can be simplified away. We also see that the moves of the value to and from the stack have disappeared through simple substitution.

$$\begin{aligned} \texttt{eax} = \; & \texttt{ValueAt(MLOC[4090f4..4090f4]}_{\texttt{const}}\texttt{)} \\ & - \ldots \\ & + (((\texttt{0xa4} \;\hat{}\; \texttt{0x25}) \mid \texttt{0x7d}) - \texttt{0x72}) \gg \texttt{0x07} \\ & + \ldots \end{aligned}$$

Figure 5.16: Unsimplified partial expression for `eax` register.

Figure 5.17 shows the results of our full analysis with simplification on the registers `eax` and `edi` at the point of the dispatch jump. Here we see that the manipulation of the value read from memory is much simpler than appeared at first. This appears to be a simple decryption where we add some immediate value, then `xor` the result with another immediate value. The value of `eax` then is just a value read from memory, and decrypted using a hard-coded decryption key. We also see that the value for `edi` is a constant value. By looking at some other instances of the dispatch routine, we see that `edi` holds the address of a table, and `eax` indexes into that table to retrieve the instruction addresses.

$$\texttt{eax}_{381} = (\texttt{ValueAt(MLOC[4090f4..4090f4]}_{\texttt{const}}\texttt{)} + \texttt{0x75}) \;\hat{}\; \texttt{0x6c}$$

$$\texttt{edi}_{381} = \texttt{0x00404200}$$

Figure 5.17: Simplified expressions for `eax` and `edi` registers.

Our system is not limited to simply analyzing hand-picked locations in the dynamic trace. Since each location typically depends on one or more previous locations in the trace, and those previous locations depend on previous locations, we found it easy to implement a general solution that performs equational reasoning to calculate a simplified expression for every single destination operand in every generated equation from the trace. Thus, examining a particular location is done by simply looking the expression up in the list of results.

As an aside, and to give an idea of how useful this automated system might be, we report on run time performance. We executed the code on a sample dynamic trace of 106407 instructions. These instructions generated a total of 261977 equations. For each equation, we ran a full expression simplification for the destination operand of each instruction in the trace. Total run time for the process averaged 489.0 seconds, corresponding to approximately 218 instructions per second or 536 equations per second. Tests were run on an Intel Core 2 Duo E6600 2.4GHz CPU with 4GB RAM.

5.3 Identifying Relevant Conditional Control Flow

As discussed in Chapter 4, value-based dependence analysis identifies the instructions that compute the values of system call arguments, but not the associated control flow instructions. The problem with identifying relevant control transfer instructions in virtualized code is that control transfers may be handled by the same dispatch code that handles other VM instructions.

In section 5.1, we introduced an equational reasoning system that is capable of handling the unique characteristics of x86 assembly code. In this section, we will use that equational reasoning system to identify instructions that implement the conditional control flow of the original program. This work is also detailed in the aforementioned CCS 2011 paper (see Coogan et al. (2011)). To do so, we use the following basic approach. First we identify all instructions in the trace that are capable of control flow (i.e., jumps, conditional jumps, calls, returns, etc...).

Next, we identify how the target addresses of those instructions are calculated using our equational reasoning tool. Any control flow instruction whose target address calculation is conditionally dependent on some previous value, is an implementation of a conditional branch statement.

To see why this statement is appropriate, we look at how conditional branches are handled on the popular IA-32 (x86) architecture – the target of our analysis tool. Conditional statements are typically implemented by setting the appropriate condition code flags in the designated flags register (eflags), e.g., using instructions such as cmp or as a side effect of arithmetic instructions such as add, then executing a conditional branch instruction such as jnz that reads this register. In essence, the target of the conditional jump is dependent on the value of the appropriate flag in the eflags register. In the case of the jnz instruction, it is dependent on the value of the "zero" flag. Each flag can only have two values – true or false. Thus, for one value of the flag, the target address will be given in the instruction, and for the other value of the flag, the target will be the address of the next instruction in the code.

It is possible that conditional logic will not be implemented exactly as described above in virtualization-obfuscated code. In fact, we have seen examples of this. VM-Protect eliminates the branch statements and moves the value of the flags register to other general purpose registers for manipulation. However, while theoretically possible, we are not aware of any obfuscation programs that implement conditional logic without the use of the value of the eflags register at some point in the code. For this reason, we begin our analysis here. We are able to identify all instructions that might set the value of the flags register, as well as its operands and how they are used.

Our equational reasoning system described in Section 5.1 translates each instruction in the dynamic trace into an equivalent set of equations. Remember that in the dynamic trace, there may be multiple equations that define the same register or memory location. Which definition is used is determined by the order that the instructions are executed in the trace. So, to maintain the original behavior of the

trace, we number the variables as follows. A variable appearing on the left hand side of an equation (i.e., a variable that is being defined) is numbered according to the order that its instruction appears in the trace. A variable appearing on the right hand side of an equation (i.e., a variable that is being used) is numbered according to the instruction that defined it. These defining instructions are found by searching backwards through the trace to identify where the definition came from.

For our purposes here, we are primarily concerned with the calculation of the target addresses of control flow instructions. Specifically, as described, we need to determine if any component of the calculation of such a target address is dependent on the value of some flag calculation. With our equational reasoning system, we need to generate a simplified expression for the target address at the point it is used, then check that expression to see if it contains any calls to the "Flag" operation.

We must also account for the possibility of additional or trivial conditional logic added for the purpose of obfuscation. The obfuscation routine cannot change the behavior of the original program, but it can add branch statements that are always true or always false, to try to confuse analysis. For this reason, we can eliminate any conditional logic that reduces to a constant Boolean value.

We will examine several examples of increasing complexity to show how our system correctly identifies these conditional dependencies. First, we look at the simple example in Figure 5.18(a), where the normal branch instructions are used to implement conditional control flow. We know that the jnz instruction uses the value of the eflags register to decide whether or not to branch, so we add a new equation at the point of the jnz instruction to represent the value of the eflags register.

As seen in Figure 5.18(b), when we trace back to find the definition of the right hand side, we see that it is the value of eflags from instruction I_{10} that is being used. By substituting the definition of eflags_{10}, we see that the value of the flags register used by the conditional jump instrution is "Flag(ebx_7 cmp eax_6)."

Next, we consider a case where the standard branch instructions are not used. This case would be anticipated when analyzing virtualization-obfuscated code, since the dispatch routine typically handles all control flow. Examining the code snippet

```
        ...
/*I10*/   cmp ebx, eax
/*I11*/   mov ebx, 0x0
/*I12*/   mov eax, 0x10
/*I13*/   jnz 10000
```

(a)

```
        ...
eflags10 = Flag(ebx7 cmp eax6)
ebx11 = 0x0
eax12 = 0x10
eflags13 = eflags10
```

(b)

Figure 5.18: Identifying control dependencies with branch instructions.

in Figure 5.19(a), we see that the indirect jump of instruction I_{16} is indexing into a table. The base of the table is at address 0x10000, and the value of eax indexes into the table some number of 4-byte values. Instruction I_{10} sets eax equal to some index value to be used. Instructions I_{11} through I_{15} perform some comparison that sets the value of the flags register, then moves the flag value into the ebx register, and masks the value. The effect is as follows. If the result of the comparison turned on the "zero" flag, then the value of the ebx register after instruction I_{14} is one. Otherwise, the value of ebx at this point is zero. The value in ebx is then added to the index value stored in eax such that the actual index value used in the jump depends on the result of the comparison in instruction I_{11}. In the context of virtualization-obfuscated code, the index value is the byte code of the next instruction, and the table contains the addresses of virtual instruction implementations. Hence, we see how conditional logic may be implemented without the use of branch statements in virtualized code.

Here, we recognize that the jump is an indirect jump, and thus depends on the target address calculation itself. We generate an equation for the calculation of the target address and insert it into our equations at the appropriate location. Next, we simplify the right hand side of our equation by substituting the definitions of the operands and simplifying the code. We repeat this process until no more substitu-

```
        ...
/*I10*/   mov eax, index
/*I11*/   cmp ebx, ecx
/*I12*/   pushf
/*I13*/   pop ebx
/*I14*/   and ebx, 0x1
/*I15*/   add eax, ebx
/*I16*/   jump [eax*4 + 0x10000]
```

(a)

```
        ...
eax10 = index
eflags11 = Flag(ebx4 cmp ecx3)
esp12 = esp9 - 4
ValueAt(MLOC[1000 .. 1003]12) = eflags11
ebx13 = ValueAt(MLOC[1000 .. 1003]12)
ebx14 = ebx13 & 0x1
eax15 = eax10 + ebx14
target16 = eax15 * 4 + 0x10000
```

(b)

Figure 5.19: Identifying control dependencies with no branch instructions.

tions or simplifications can be performed. The end result is given in Figure 5.20. We note that the resulting expression depends on the result of the "Flag" operation, and thus the indirect jump is acting as a conditional control flow statement.

$$\mathsf{target}_{16} = \mathsf{index} + (\mathsf{Flag}(\mathsf{ebx}_4\ \mathsf{cmp}\ \mathsf{ecx}_3)\ \&\ \mathsf{0x1})$$

Figure 5.20: Result of target address simplification from Figure 5.19.

Finally, we examine a case inspired by the conditional control flow implementation that is used in VMProtect. We begin with the example in Figure 5.19, and add the use of indirection. In the code snippet of Figure 5.21(a), two index values are stored to adjacent locations in memory. The same trick is then used to conditionally load either the address of the first index or the address of the second index into the esi register. Finally, the value stored at the location in esi is loaded into the eax register, and the indirect jump calculates the address in the table to use.

```
          ...
/*I10*/    mov [address], index1
/*I11*/    mov [address + 4], index2
/*I12*/    mov esi, address
/*I13*/    cmp ebx, ecx
/*I14*/    pushf
/*I15*/    pop ebx
/*I16*/    and ebx, 0x1
/*I17*/    mul ebx, 0x4
/*I18*/    add esi, ebx
/*I19*/    mov eax, [esi]
/*I20*/    jump [eax*4 + 0x10000]
```

(a)

```
          ...
ValueAt(MLOC[address .. address + 3]10) = index1
ValueAt(MLOC[address + 4 .. address + 7]11) = index2
esi12 = address
eflags13 = Flag(ebx4 cmp ecx3)
esp14 = esp9 - 4
ValueAt(MLOC[1000 .. 1003]14) = eflags13
ebx15 = ValueAt(MLOC[1000 .. 1003]14)
esp15 = esp14 + 4
ebx16 = ebx15 & 0x1
ebx17 = ebx16 * 0x4
esi18 = esi12 + ebx17
eax19 = ValueAt(esi18)
target20 = eax19 * 4 + 0x10000
```

(b)

Figure 5.21: Example of code using indirection to hide control dependencies.

For this example, let's assume that it was *index1* that was used in the calculation of the target address, and that the value of *address* was 5000. When we simplify our equation for target_{20}, we will substitute "ValueAt(esi_{18})" for "eax_{19}." By our assumption, we know that esi_{18} holds the value 5000, so we substitute the value at memory location 5000 which is *index1* from instruction I_{10}. Our simplified expression for the target is then "target_{20} = *index1* * 4 + 0x10000." From this result, it appears that this jump is not implementing conditional control flow. However, we know this to be wrong. We know from our analysis of the code that the result of the compare instruction determines whether *index1* or *index2* is used to index into the table. The problem here, is that the conditional element has been hidden by a layer of indirection. When we calculate an expression for the target address, we are only using direct dependencies.

$$
\begin{array}{ll}
 & \ldots \\
 & \text{ValueAt}(\texttt{MLOC}[5000 \,..\, 5003]_{10}) = \textit{index1} \\
 & \text{ValueAt}(\texttt{MLOC}[5004 \,..\, 5007]_{11}) = \textit{index2} \\
 & \text{esi}_{12} = 5000 \\
 & \text{eflags}_{13} = \text{Flag}(\text{ebx}_4 \text{ cmp } \text{ecx}_3) \\
 & \text{esp}_{14} = \text{esp}_9 - 4 \\
 & \text{ValueAt}(\texttt{MLOC}[1000 \,..\, 1003]_{14}) = \text{eflags}_{13} \\
 & \text{ebx}_{15} = \text{ValueAt}(\texttt{MLOC}[1000 \,..\, 1003]_{14}) \\
 & \text{esp}_{15} = \text{esp}_{14} + 4 \\
 & \text{ebx}_{16} = \text{ebx}_{15} \,\&\, 0\text{x}1 \\
 & \text{ebx}_{17} = \text{ebx}_{16} * 0\text{x}4 \\
 & \text{esi}_{18} = \text{esi}_{12} + \text{ebx}_{17} \\
(*) & \texttt{MLOC}[5000 \,..\, 5003]_{19} = \text{esi}_{18} \\
 & \text{eax}_{19} = \text{ValueAt}(\texttt{MLOC}[5000 \,..\, 5003]_{19}) \\
 & \text{target}_{20} = \text{eax}_{19} * 4 + 0\text{x}10000
\end{array}
$$

Figure 5.22: Equations augmented to handle indirection.

To resolve this issue, we must account for any number of layers of indirection. We could do this by analyzing the code, as we did here, and recognizing that it was address 5000 that was used, and not 5004. To do this, however, we would need to understand how the code works at each step. As the code becomes more complex, this task becomes harder and more time consuming. Instead, we do this by using

the memory location equations that we described in Section 5.1. Remember that we introduce a new variable for each memory access. The name of this new variable is the prefix "MLOC" followed by the actual address range that was accessed. Then a new equation is added that sets the value of that new variable according to how the calculation was done. Each variable is labeled with the order number as before to guarantee that they are unique. Figure 5.22 shows the addition of the memory location equation (marked with a *) for the memory access of instruction I_{19} in Figure 5.21. In practice, we would add similar equations for all memory accesses, but these are not shown here for the sake of clarity.

Now, for each memory access that is used to calculate the target address, we can start a new simplified expression for the address calculation. If this calculation shows some dependency on the "Flag" operation, then we know that the target address is indirectly conditionally dependent. Any memory access simplified expression that does not show any conditional dependence can be discarded because it is irrelevant. Returning to our example in Figure 5.21, we calculate all simplified expressions for memory access and get the results shown in Figure 5.23. Here we see the conditional dependence that we expect, and can mark the indirect jump of instruction I_{20} as implementing conditional control flow.

$$\texttt{MLOC}[10000 \,..\, 10003]_{19} = \mathsf{address} + (\mathsf{Flag}(\mathsf{ebx}_4 \ \mathsf{cmp}\ \mathsf{ecx}_3)\ \&\ 0\mathsf{x}1)$$
$$\mathsf{target}_{20} = \mathit{index1} * 4 + 0\mathsf{x}10000$$

Figure 5.23: Result of target address simplification from Figure 5.21.

To handle multiple layers of indirection, we also add in a new simplified expression calculation for each memory access used to calculate these address calculations, and so on. A naive approach to this problem would look backwards through the code, adding new simplified expressions recursively as they were needed. This approach is time consuming, and can result in much repeated or unnecessary work. Instead, we have implemented an algorithm that propagates dependency information forward through the code at each step. Conditional control flow can then be identified by

looking to see if any dependencies are associated with the current instruction from previous calculations.

The algorithm presented in Figure 5.24 assumes that the equivalent equations for each instruction in the dynamic trace have been created. Then, for each equation, in trace order, it determines if that equation has a direct conditional dependency, and adds itself to the list of dependencies if it does. Next, it adds in all of the dependencies of its memory access equations. This is how indirect dependencies are collected. Then the simplified version of the equation is created by substituting term definitions for its operands then simplifying the resulting expression. This process continues until no more substitutions can be made and no more simplifications are possible. At each step, the conditional dependencies of each term replacement are saved and added to the list as well. As a final step, the list of dependencies is associated with that equation. Later the list will be searched, and the appropriate instruction marked if there is a non-trivial conditional dependency.

5.4 Experimental Results

We have added the results of our conditional control flow to those presented in Section 4.4. Table 5.1 shows the relevant and obfuscation percentages for our VM-Protect protected test files when the conditional control flow is included. Similarly, Table 5.2 shows the relevant and obfuscation percentages for our CodeVirtualizer protected test files when the conditional control flow is included.

Table 5.1: Results for programs obfuscated with VMProtect

Name	Original trace size	relevant matching	Rel. Score	Obf. Score
factorial	92	52	56.5%	99.0%
matrx_mult	651	344	52.8%	99.9%
fibonacci	151	55	36.4%	99.5%
BullMoose	94	36	38.3%	95.1%
hunatcha	2226	1333	59.9%	119.5%
md5	2257	1687	74.8%	99.2%

```
Input: List of Equations : EqnList
Output: List of simplified expressions : SimpEqnList

Deps = ∅
SimpEqnList = ∅
for each equation in EqnList:
    if IsConditional(equation)
    then
      Deps = Deps ∪ equation
    endif
    MemAccesses = GetMemoryAccessesForEqn(equation)
    for each memAcc in MemAccesses:
      Deps = Deps ∪ memAcc→deps
    endfor
    ExprList = SimpEqnList ∪ equation
    for each term in equation:
      replacement = FindReplacement(EqnList, term)
      if replacement != ∅
      then
        ReplaceTerm(term, replacement)
        Deps = Deps ∪ replacement→deps
      endif
      equation→deps = Deps
    endfor
endfor
return SimpEqnList
```

Figure 5.24: Algorithm for tracking all conditional dependencies.

One interesting data point is the overly high obfuscation score of 119.5% for the VMProtected version of the "hunatcha" virus. This value is actually a product of our less than perfect approach to measuring the effectiveness of our algorithm. With the obfuscation score, we are trying to calculate the percentage of obfuscation instructions that were correctly identified and removed. We know how many instructions are eliminated from the virtualized trace. However, the problem is that we don't know which instructions are actually obfuscation instructions. Thus, we approximate the obfuscation instructions as all of the instructions that are removed. We know, from the fact that the relevant score is not 100%, that this actually includes obfuscation instructions plus original program instructions that were not

Table 5.2: Results for programs obfuscated with CodeVirtualizer

Name	Original trace size	relevant matching	Rel Score	Obf Score
factorial	92	54	58.7%	71.7%
matrx_mult	651	452	69.4%	82.8%
fibonacci	151	99	65.6%	91.7%
BullMoose	94	66	70.2%	72.9%
hunatcha	2226	1524	68.5%	72.5%
md5	2257	2098	93.0%	54.4%

correctly identified. Hence, obfuscation scores over 100% simply represent the case where these two values combined are greater than the total number of obfuscation instructions.

Overall, the addition of conditional control flow has a small effect on the relevant and obfuscation scores. However, it is a first step in a larger attempt to recreate the original code. We have argued that the results of Chapter 4 give us an understanding into the behavior of a single execution trace. The inclusion of conditional control flow information should eventually provide the ability to reconstruct an approximation of the static control flow graph.

CHAPTER 6

CONCLUSIONS

Obfuscation techniques have gained popularity in recent years in the context of computer security. Obfuscation techniques are often used to protect copyrighted or trade secret algorithms in commercial products, as well as to prevent tampering with license verification to prevent illegal use of that software. However, they have also become popular with the authors of malware as a means of avoiding detection and preventing reverse-engineering. When used in this way, obfuscation techniques are a security threat to legitimate users, and there is a need to understand and defeat such techniques.

The obfuscation technique known as "packing" scrambles the bytes of an executable file so that static analysis techniques are rendered ineffective. Dynamic techniques, however, are vulnerable to defenses specifically designed to thwart dynamic analyses. We present an alternative approach that uses well-understood static analysis techniques to identify the unpacking code that comes with a given malware binary, then uses this code to construct a customized unpacker for that binary. This customized unpacker can then be executed or emulated to obtain the unpacked malware code. Our approach does not presuppose any knowledge about the software or algorithm used to create the packed binary. Preliminary results from a prototype implementation suggest that our approach can effectively unpack a variety of packed malware, including some constructed using custom packers and some obtained using commercial binary packing tools.

While this success is encouraging, it also suffers from certain shortcomings that may not be able to be resolved. Most importantly, the static analysis that we use relies heavily on the precision of the pointer analysis that we implemented. In the case of malware analysis, this will be an issue going forward. We must assume that malware authors will know what analysis techniques we are using. Since the

problem of pointer analysis is known to be undecidable in the general case, it should be simple matter for these authors to include sufficient pointer arithmetic in their protection schemes so as to make our analysis impractical. That is, our tool will still work in theory, but run-time limitations will prevent useful results. We currently do not see a workaround for this problem.

These problems with static analysis suggest that dynamic analysis techniques should be more useful in the context of malware. As stated before, the problem of unpacking packed executables has been studied (see Martignoni et al. (2007) and Kang et al. (2007)). However, there are multiple other obfuscation techniques that have become popular in recent years, such as virtualization-obfuscation.

Similar to packed malware, virtualization-obfuscated protected programs are difficult to reverse engineer because examining the machine instructions of the program, either statically or dynamically, does little to shed light on the program's logic, which is hidden in the interpreted byte-code. Prior approaches to reverse-engineering virtualization-obfuscated programs typically work by first reverse engineering the byte-code interpreter, and then working back from this to work out the logic embedded in the byte code. We present a different approach that focuses on identifying the flow of values to system call instructions. This new approach can be applied to a larger number of obfuscated binaries because it makes fewer assumptions about the nature of the virtual machine interpreter.

Our current implementation uses a dynamic trace to identify those instructions that effect the value of system call parameters, thus identifying the instructions that are relevant to the program's interaction with the system. It also identifies branch instructions by identifying dependencies between the calculation of the target addresses of control flow instructions and conditional logic in the code.

The system works by gradually adding instructions that are calculated to be of importance, and adding these to a relevant subtrace that represents the behavior of the original, unobfuscated program. Experimental results of several programs including two malware executable files and one benchmark utility show good results. Many instructions labeled by the analysis as relevant instructions are indeed part of

the original, unobfuscated program. In addition, our system does a very good job of avoiding false positives, and not including instructions that only function as part of the virtual machine interpreter. We believe that such a system will be a useful tool to complement existing techniques that deobfuscate virtualized malware.

The work presented in this dissertation is intended to be an early step toward a new approach to malware. The "arms race" that has evolved over the years in the area of malware detection has left security analysts trailing in the fight to control the damage caused by malware. While our work deals with two separate types of obfuscation, we have been consistent in applying a new behavior-based approach that, in theory, should be able to deal with new instances of these techniques. It should be easy to adapt to new, unforeseen techniques for protecting malicious code.

REFERENCES

(2011). VX Heavens. `http://vx.netlux.org/`.

Aho, A. V., R. Sethi, and J. D. Ullman (1985). *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

ASPack Software (2009). ASPack Software. Http://www.aspack.com/asprotect.aspx.

Balakrishnan, G. (2007). *WYSINWYX: What You See Is Not What You eXecute*. Ph.D. thesis, Computer Science Department, University of Wisconsin, Madison.

Balakrishnan, G., R. Gruian, T. W. Reps, and T. Teitelbaum (2005). CodeSurfer/x86-A Platform for Analyzing x86 Executables. In Bodk, R. (ed.) *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3443 of *Lecture Notes in Computer Science*, pp. 250–254. Springer. ISBN 3-540-25411-0. doi:http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3443&spage=250.

Balakrishnan, G. and T. Reps (2004). Analyzing memory accesses in x86 executables. In *Proc. 13th. International Conference on Compiler Construction*, pp. 5–23.

Bayer, U., P. M. Comparetti, C. Hlauschek, C. Krgel, and E. Kirda (2009). Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium*.

Bellard, F. (2005). QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46. USENIX.

Bergeron, J., M. Debbabi, M. M. Erhioui, and B. Ktari (1999). Static analysis of binary code to isolate malicious behaviors. In *Proc. IEEE 8th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '99)*, pp. 184–189.

Black Fenix (????). Black Fenix's Anti-Debugging Tricks. `http://members.fortunecity.com/blackfenix/isbpxede.html`.

Brauer, J., B. Schlich, T. Reinbacher, and S. Kowalewski (2009). Stack bounds analysis for microcontroller assembly code. In *Proceedings of the 4th Workshop on Embedded Systems Security*, WESS '09, pp. 5:1–5:9. ISBN 978-1-60558-700-4.

Cesare, S. (1999). Linux anti-debugging techniques (fooling the debugger). VX Heavens.
`http://vx.netlux.org/lib/vsc04.html`.

Chang, H. and M. Atallah (2001). Protecting Software Code by Guards. In *Security and Privacy in Digital Rights Management, ACM CCS-801, Philadelphia, PA, USA, November 5, 2001, Revised Papers*, pp. 160–175.

Christodorescu, M. and S. Jha (2003). Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pp. 12–12. USENIX Association, Berkeley, CA, USA.

Coen-Porisini, A., G. Denaro, C. Ghezzi, and M. Pezzé (2001). Using symbolic execution for verifying safety-critical systems. In *Proceedings of the 8th European software engineering*, ESEC/FSE-9, pp. 142–151. ISBN 1-58113-390-1.

Collberg, C., C. Thomborson, and D. Low (1997). A Taxonomy of Obfuscating Transformations.

Collberg, C. S. and C. Thomborson (2002). Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *Software Engineering, IEEE Transactions on*, **28**, pp. 735–746.

Coogan, K. and S. Debray (2011). Equational Reasoning on x86 Assembly Code. *Source Code Analysis and Manipulation, IEEE International Workshop on*.

Coogan, K., S. Debray, T. Kaochar, and G. Townsend (2009). Automatic Static Unpacking of Malware Binaries. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pp. 167–176. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3867-9. doi:http://dx.doi.org/10.1109/WCRE.2009.24.

Coogan, K., G. Lu, and S. Debray (2011). Deobfuscating Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11. ACM, New York, NY, USA.

Danielescu, A. (2008). Anti-debugging and Anti-emulation Techniques. *CodeBreakers Journal*, **5**(1). `http://www.codebreakers-journal.com/`.

Dinaburg, A., P. Royal, M. I. Sharif, and W. Lee (2008). Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pp. 51–62.

Djoudi, L. and L. Kloul (2008). Assembly Code Analysis Using Stochastic Process Algebra. In *Proceedings of the 5th European Performance Engineering Workshop on Computer Performance Engineering*, EPEW '08, pp. 95–109. ISBN 978-3-540-87411-9.

Falliere, N. (2009). Inside the Jaws of Trojan.Clampi. Technical report, Symantec Corp.

Ferrie, P. (2008). Prophet and Loss. *Virus Bulletin*. `www.virusbtn.com`.

Gill, A. (2006). Introducing the Haskell equational reasoning assistant. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pp. 108–109. ISBN 1-59593-489-8.

Harman, M. and S. Danicic (1998). A new Algorithm for slicing unstructured programs. *Journal of Software Maintenance*, **10**(6), pp. 415–441.

Hind, M. and A. Pioli (2000). Which pointer analysis should I use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 113–123.

Hypponen, M. (2007). Mobile Malware. Invited talk, USENIX Security '07.

Intel Corp. (2011). *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corp.

Jones, N. D., C. K. Gomard, and P. Sestoft (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

Jorge, J. S., V. M. Gulias, and J. L. Freire (2009). Certifying properties of an efficient functional program for computing Gröbner bases. *J. Symb. Comput.*, **44**, pp. 571–582. ISSN 0747-7171.

Julus, L. (1999). Anti-Debugging in Win32. VX Heavens. `http://vx.netlux.org/lib/vlj05.html`.

Kang, M. G., P. Poosankam, and H. Yin (2007). Renovo: A Hidden Code Extractor for Packed Executables. In *Proc. Fifth ACM Workshop on Recurring Malcode (WORM 2007)*.

Kinder, J. and H. Veith (2008). Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pp. 423–427. ISBN 978-3-540-70543-7.

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, **19**, pp. 385–394. ISSN 0001-0782.

Kruegel, C., W. Robertson, F. Valeur, and G. Vigna (2004). Static disassembly of obfuscated binaries. In *In Proceedings of USENIX Security*, pp. 255–270.

Kwon, T. and Z. Su (2010). Modeling High-Level Behavior Patterns for Precise Similarity Analysis of Software. `www.cs.ucdavis.edu/research/tech-reports/2010/CSE-2010-16.pdf`.

Landi, W. (1992). Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, **1**, pp. 323–337.

Lawton, G. (2009). On the Trail of the Conficker Worm. *Computer*, **42**(6), pp. 19 –22. ISSN 0018-9162. doi:10.1109/MC.2009.198.

Lee, T. and J. J. Mody (2006). Behavioral Classification. In *EICAR Conference*.

Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, **52**, pp. 107–115. ISSN 0001-0782.

Linn, C. and S. Debray (2003). Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pp. 290–299. ACM, New York, NY, USA. ISBN 1-58113-738-9. doi:http://doi.acm.org/10.1145/948109.948149.

Low, D. (1998). Protecting Java code via code obfuscation. *Crossroads*, **4**, pp. 21–23. ISSN 1528-4972. doi:http://doi.acm.org/10.1145/332084.332092.

Maebe, J. and K. De Bosschere (2003). Instrumenting self-modifying code. In *Proc. Fifth International Workshop on Automated Debugging (AADEBUG2003)*, pp. 103–113.

Mahadik, V. A. (????). Reverse Engineering of the Honeynets SOTM32 Malware Binary. The Honeynet Project, `http://www.honeynet.org/scans/scan32/sols/4-Vinay_A_Mahadik/Analysis.htm`.

Mahlke, S. A., D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann (1992). Effective compiler support for predicated execution using the hyperblock. *SIGMICRO Newsl.*, **23**, pp. 45–54. ISSN 1050-916X. doi:http://doi.acm.org/10.1145/144965.144998.

Martignoni, L., M. Christodorescu, and S. Jha (2007). OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 21st Annual Computer Security Applications Conference, ACSAC 2007, Miami Beach, Florida, USA*. IEEE Computer Society.

Moore, D., V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver (2003). Inside the Slammer Worm. *IEEE Security and Privacy*, **1**(4).

Myreen, M. O. (2009). Formal verification of machine-code programs.

Oberhumer, M. F. X. J., L. Molnár, and J. F. Reiser (2008). UPX: the Ultimate Packer for eXecutables. Http://upx.sourceforge.net/.

Oreans Technologies (2008). Code Virtualizer: Total Obfuscation Against Reverse Engineering. `http://www.oreans.com/codevirtualizer.php`.

PEid (2008). PEid. Http://www.peid.info/.

Preda, M. D. and R. Giacobazzi (2005). Semantic-based code obfuscation by abstract interpretation. In *In Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP05*, pp. 1325–1336. Springer.

Ramalingam, G. (1994). The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, **16**, pp. 1467–1471. ISSN 0164-0925. doi:http://doi.acm.org/10.1145/186025.186041.

Rayside, D. (2005). Points-To Analysis. MIT OpenCourseWare, Massachusetts Institute of Technology. `http://ocw.mit.edu/`.

Rogue Warrior (1996). Guide to Improving Polymorphic Engines. `http://vx.netlux.org/lib/vrw02.html`.

Rolles, R. (2009). Unpacking Virtualization Obfuscators. In *Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT '09)*.

Russinovich, M. and D. Solomon (2005). *Microsoft Windows internals: Microsoft Windows server 2003, Windows XP, and Windows 2000*. Pro-Developer. Microsoft Press. ISBN 9780735619173.

Sharif, M., A. Lanzi, J. Giffin, and W. Lee (2009). Automatic Reverse Engineering of Malware Emulators. In *Proc. 2009 IEEE Symposium on Security and Privacy*.

Sharif, M. I., A. Lanzi, J. T. Giffin, and W. Lee (2008). Impeding Malware Analysis Using Conditional Code Obfuscation. In *Network and Distributed System Security Symposium*.

Song, Y., M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo (2007). On the infeasibility of modeling polymorphic shellcode. In *Computer and Communications Security*, pp. 541–551. doi:10.1145/1315245.1315312.

Szor, P. (2005). *The Art of Computer Virus Research and Defense*. Symantek Press.

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, **3**, pp. 121–189.

Udupa, S. K., S. K. Debray, and M. Madou (2005). Deobfuscation: Reverse Engineering Obfuscated Code. *Reverse Engineering, Working Conference on*, **0**, pp. 45–54. ISSN 1095-1350. doi:http://doi.ieeecomputersociety.org/10.1109/WCRE.2005.13.

Venkitaraman, R. and G. Gupta (2004). Static program analysis of embedded executable assembly code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '04, pp. 157–166. ISBN 1-58113-890-3.

VMProtect Software (2008). VMProtect Software Protection. http://vmpsoft.com/.

Walenstein, A., R. Mathur, M. R. Chouchane, and A. Lakhotia (2006). Normalizing Metamorphic Malware Using Term Rewriting. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 75–84. ISBN 0-7695-2353-6.

Wang, C., J. Hill, J. C. Knight, and J. W. Davidson (2001). Protection of Software-Based Survivability Mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, DSN '01, pp. 193–202. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-1101-5.

Wells, J. B. and R. Vestergaard (2000). Equational Reasoning for Linking with First-Class Primitive Modules. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pp. 412–428. ISBN 3-540-67262-1.

Wroblewski, G. (2002). General Method of Program Code Obfuscation.

Xu, B., J. Qian, X. Zhang, Z. Wu, and L. Chen (2005). A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, **30**(2), pp. 1–36.