TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

Analyzing Malicious Code and Infrastructure

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

Privatdozent Dipl.-Ing. Dr.techn. Christopher Krügel
Department of Computer Science
University of California, Santa Barbara

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Andreas Moser
Matrikelnummer: 9926603
Karmarschgasse 51/1/25
A-1100 Wien

Wien, am 17. Dezember 2009

Kurzfasung (German Abstract)

Im Laufe der letzten Jahre hat sich die Anzahl der im Internet vorhandenen Schadsoftware vervielfacht. Hauptsächlich verantwortlich ist auf der einen Seite schlicht die gestiegene Popularität des Internets und die damit einhergehende erhöhte Anzahl an lohnenden Zielen. Ein weiterer Grund dafür ist jedoch auch, dass es Angreifern immer öfter gelingt, mit Hilfe kompromittierter Computer enorme Gewinne zu erzielen. Die Aussicht auf diese Profite motiviert immer öfter gut organisierte, kriminelle Gruppen sich an der Verbreitung von Schadprogrammen aktiv zu beteiligen. Die bösartige Software, die von diesen, an Profit orientierten, Gruppen verwendet wird, weißt dabei eine weit größere Professionalität auf als bisher bekannte Schadprogramme. Im Rahmen dieser Arbeit sollen nun innovative Möglichkeiten aufgezeigt werden, die es erlauben, auch solche neuartige Schadsoftware effizient mittels automatischer Werkzeuge zu analysieren.

Zuerst wird gezeigt, dass für die Analyse von Software, die bewusst Gegenmaßnahmen einsetzt um ebendiese Analyse zu verhindern, so genannte dynamische Analyseverfahren zu verwenden sind. Es werden Verfahren vorgestellt, die mit wenig Aufwand Schadcode so modifizieren, dass herkömmliche statische Analyseverfahren nicht mehr in der Lage sind, aufschlussreiche Informationen über die analysierte Software zu generieren. Da dynamische Analysemethoden gegen die hier vorgestellten Verfahren immun sind, sollte ihnen der Vorzug gegenüber statischen Methoden zur Schadcodeanalyse gegeben werden.

Ein Nachteil, den dynamische Analysemethoden im Gegensatz zu statischen Ansätzen jedoch besitzen, ist die geringere Abdeckung des Programmcodes eines analysierten Programms. Dadurch können unter Umständen Schadprogramme, die ihr bösartiges Verhalten nur unter sehr speziellen Bedingungen preisgeben, bei einer dynamischen Analyse fälschlich als gutartig eingestuft werden. Um diesem

Problem entgegenzuwirken, wird in dieser Arbeit ein Ansatz vorgestellt, der dynamische Analyseverfahren erweitert. Es werden dabei Programmteile, die nur unter bestimmten Bedingungen, die während einer Analyse normalerweise nicht zutreffen, ausgeführt und analysiert.

Um dem Problem der Schadsoftware noch effizienter zu begegnen, als es mit den hier vorgestellten Analysemethoden möglich ist, wird auch der Prototyp eines Netzwerkanalysesystems vorgestellt. Dieses System ist in der Lage, Netzwerke aufzuspüren, die von Angreifern verwendet werden, um Schadsoftware in Umlauf zu bringen sowie andere Internet-basierte Angriffe zu starten. Mit Hilfe dieses Ansatzes ist es möglich, solche Netzwerke frühzeitig zu erkennen und bloßzustellen, was im besten Fall dazu führen kann, dass bösartige Netzwerke vom Internet abgeschnitten werden. Dadurch kann die Infrastruktur etwaiger Angreifer derart gestört werden, dass die Auslieferung von Schadcode an ihre Opfer fehlschlägt, wodurch die Sicherheit für Internetbenutzer immens gesteigert wird.

Abstract

During the past few years, the damage caused by malware has dramatically increased. One reason is the rising popularity of the Internet and the resulting increase in the number of available vulnerable machines because of security-unaware users. Another reason is the elevated sophistication of the malicious code itself. Organized groups are releasing highly professional malicious software that contains sophisticated anti-analysis methods and payload for monetary gains. This fact brings up new, challenging problems for analysts of malicious software. In this work, we will present methods that allow to counter this threat. We will show both how current malicious software can be effectively analyzed by automated tools and how the underlying infrastructure of networks distributing malicious code can be exposed.

First, we show that for analyzing malicious code that deliberately tries to prevent analysis, dynamic analysis methods have to be preferred over static methods. We present a system that can easily transform malicious binaries into semantically equivalent programs such that static analysis methods can no longer gather any useful information from the binary. The only way to analyze those resulting programs is to use dynamic analysis, which is immune to the applied transformations.

Even though dynamic analysis methods are better suited to analyze malicious software, they usually have the problem of lower code coverage during the analysis than static methods, which usually analyze the complete binary. This can lead to the misclassification of a malicious program as benign if it executes its malicious payload only under very specific circumstances. To mitigate this problem, we present an approach to extend current dynamic analysis tools in a way that covers also code paths that are executed only if very specific conditions are met.

Finally, we present a system that is able to identify networks used by miscreants to distribute malicious code and launch other attacks against Internet users. By exposing those malicious networks, it is possible to generate effective blacklists that disallow access to servers hosted there or to even disconnect the whole network from the Internet by other Internet service providers. In both cases, the distribution of malicious programs to the victims of exploits can be prevented.

Acknowledgments

I owe my thanks to my advisors, Christopher Kruegel and Engin Kirda, who have safely and skillfully guided me through the entirety of the work presented in this thesis. They have earned my gratitude and respect with their professional competence, their catching enthusiasm, and the admirable ability to keep their students work steadily on the road of science. Their positive example has strongly influenced my way of working, and has resulted in a dissertation that I believe to be both interesting and nice to read.

Thanks go also to my colleagues at the Vienna University of Technology. In particular, I want to express my gratitude for the fruitful discussions we had, the interesting collaborations that were made possible and the good working environment provided by them.

The final version of this work was created during a stay at the Secure Systems Lab of the University of California, Santa Barbara. I'd like to thank the members of the lab for the interesting insights I gained during our discussions and for making my stay as pleasurable as it was.

Finally, I want to thank all my friends for the wonderful time we spent together and, last but not least, my parents for their support during all those years that made this dissertation possible.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Outline 2

2 Analysis of Malicious Code **5**

2.1 Static Analysis Techniques 6

2.2 Dynamic Analysis Techniques 9

3 Limits of Static Analysis for Malware Detection **13**

3.1 Code Obfuscation 16

3.1.1 Opaque Constants 16

3.1.2 Obfuscating Transformations 21

3.2 Binary Transformation 24

3.3 Evaluation 26

3.3.1 Evasion Capabilities 26

3.3.2 Transformation Robustness 30

3.3.3 Size and Performance 32

3.3.4 Possible Countermeasures 33

3.3.5 Summary 33

4 Exploring Multiple Execution Paths for Malware Analysis **35**

4.1 System Overview 38

4.2 Path Exploration 41

4.2.1 Tracking Input 42

4.2.2 Saving and Restoring Program State 48

4.3 System Implementation 50
4.3.1 Creating and Restoring Program Snapshots 51
4.3.2 Identification of Program Termination 53
4.3.3 Optimization 54
4.3.4 Limitations 54
4.4 Evaluation 56
4.5 Summary 63

5 Finding Rogue Networks **65**
5.1 System Overview 68
5.2 Data Collection 71
5.2.1 Botnet Command and Control Providers 71
5.2.2 Drive-by-Download Hosting Providers 73
5.2.3 Phish Hosting Providers 75
5.3 Data Analysis 76
5.3.1 Longevity of Malicious IP Addresses 76
5.3.2 Malscore Computation 78
5.4 Evaluation 80
5.4.1 Analysis Results and Malicious Networks 80
5.4.2 Interesting (Historic) Malscore Changes 83
5.4.3 Sensitivity of Important Parameters 85
5.5 Summary 87

6 Related Work **91**

7 Conclusions **99**

List of Tables

3.1 Evasion results for four commercial virus scanners 27

4.1 Number of samples that access tainted input sources. 58
4.2 Relative increase of code coverage. 59

5.1 *FIRE* Top 10 for June 1st, 2009 81
5.2 ShadowServer Botnets Top 10 for June 1st, 2009 82
5.3 Google Safe Browsing Top 10 for June 1st, 2009 82
5.4 ZeusTracker Top 10 for June 1st, 2009 83

List of Figures

3.1 Opaque constant calculation 17
3.2 Opaque constant based on 3SAT 19

4.1 Exploration of multiple execution paths. 40
4.2 Consistent memory updates. 44
4.3 Constraints generated during program execution. 47
4.4 `Blaster` source code snippet. 60
4.5 `rxBot` source code snippet. 61

5.1 Architecture for C&C botnet monitoring 73
5.2 Uptimes for different sources 77
5.3 De-peering effects. 84
5.4 Sensitivity of Parameter c. 86
5.5 Ranking changes for varying thresholds. 89

Chapter 1

Introduction

1.1 Motivation

Malicious code (or malware) is defined as software that fulfills the deliberately harmful intent of an attacker. Nowadays, such software poses a major security threat to computer users. According to estimates, the financial loss caused by malware has been as high as 14.2 billion US dollars in the year 2005 [24]. Unfortunately, the problem of malicious code is likely to grow in the future as malware writing is quickly turning into a profitable business [100]. Malware authors can sell their creations to miscreants, who use the malicious code to compromise large numbers of machines that can then be abused as platforms to launch denial-of-service attacks or as spam relays. Another indication of the significance of the problem is that even people without any special interest in computers are aware of worms such as Storm [88] or Conficker [103]. This is because security incidents affect millions of users and regularly make the headlines of mainstream news sources.

The traditional line of defense against malware is composed of malware detectors such as virus and spyware scanners. Unfortunately, both researchers and malware authors have demonstrated that these scanners, which use pattern matching to identify malware, can be easily evaded. To address this shortcoming, more powerful malware detectors have been proposed. Many of these tools rely on semantic signatures and employ static analysis techniques such as model checking

and theorem proving to perform detection. While it has been shown that these systems are highly effective in identifying current malware, it is less clear how successful they would be against adversaries that take into account the novel detection mechanisms. In this thesis, we will present program obfuscation methods that can be applied directly to binaries and can effectively thwart detection of malicious code by conventional virus scanners and also advanced malware detection tools.

Another class of malware detectors apply dynamic analysis methods to identify malware binaries. In this approaches, the binary to analyze is executed in a restricted environment and all of its interactions with the operating system are recorded. Later, the trace of these interactions can be inspected and evidence of suspicious activities like unwanted network connections or other malicious behavior can be used to classify a binary as malicious.

While these dynamic analysis systems usually work really well with current malware binaries, there are some examples where malicious activity that is dormant in an executable is missed by the analysis system. This is due to the fact that in the simulated environment external events the binary is waiting for (for example a command sent over an IRC channel by the owner of a bot network) are simply not present. In this thesis, we will present novel methods that deal with those trigger-based malware samples and can reliably reveal this hidden, malicious behavior.

Another approach that we want to show in this thesis to counter malware is to identify the networks that are used by miscreants to distribute their malicious software. If we can find a way to block access to those networks or, even better, to disconnect those malicious networks completely from the Internet, many of the malware infections we see taking place every day could be prevented.

1.2 Thesis Outline

In this thesis we will present new methods to dynamically analyze malicious binaries and also a network monitoring tool that can be used to detect malware distributing networks. This thesis is based on my previously published work [9, 70, 71, 99].

In Chapter 2 we will give an overview on techniques that are currently used to analyze malicious code. We will show the different methods available as well as their respective strengths and weaknesses.

In Chapter 3 we show, how traditional virus scanners as well as more sophisticated static analysis tools can be fooled by applying certain binary transformations to malware samples. We show how these transformations will easily thwart detection by all of the traditional malware scanners we tested which gives a strong motivation to concentrate on dynamic analysis methods as those methods are completely unaffected by the presented transformations.

To address the problem of trigger-based malware samples that could be misclassified by conventional dynamic analysis systems, we present an extension to those systems in Chapter 4. The presented method can be used to reveal the hidden malicious behavior present in the analyzed malware samples as well as the external events that would lead to those malicious actions.

However, even if we can accurately assess if a binary is malicious or not using the presented tools, we think it would be much better if we could get to the root of the problem of malware and hinder the distribution of malicious binaries. Thus, in Chapter 5 we present a network scanning framework that is able to detect rogue, malware distributing networks and to generate blacklists for the identified hosts.

Chapter 2

Analysis of Malicious Code

Malware analysis is the process of determining the purpose and functionality of a given sample of malicious code. This process is a necessary step to be able to develop effective detection techniques for malicious software. Nowadays, the most important line of defense against malicious code are still virus scanners. These rather unintelligent search programs typically scan all files they encounter for predefined binary strings (so called *signatures*). As soon as the signature of a known malicious code sample is found, an alarm is raised and the user is informed that his computer is infected by malicious code. The huge drawback of this approach is that virus scanners rely on a database of signatures that characterize known malware instances. Thus, only known malicious code samples that are already present in the database can be found by the anti virus software. Every time a new malware sample is found in the wild, there is a window of vulnerability in which the virus scanner has not updated its signature database and, thus, is not able to raise an alarm if it encounters this malware sample. Therefore, it is extremely important to analyze newly found unknown software samples as fast as possible in order to timely update the signature databases if the binary is found to be malicious. However, realizing that a given binary sample is malicious is often not sufficient. Frequently, it is necessary to gain a detailed understanding of the behavior of the program. For example to be able to cleanly remove a piece of malware from an infected machine, it is usually not enough to delete the binary itself. It is also necessary to remove the residues left behind by the malicious

code (such as unwanted registry entries, services, or processes) and undo changes made to legitimate files. Also, in cases of bot clients, it is important to understand the mechanisms behind the corresponding bot network. This information can lead to much better defense measures than the protection offered by virus scanners. For example by revealing a bots communication patterns, the command and control channels of the whole bot network can be undermined and subsequently the responsible servers can be shut down.

The traditional approach to analyze the behavior of an unknown program is to execute the binary in a restricted environment and observe its actions. The restricted environment is often a debugger, used by a human analyst to step through the code in order to understand its functionality. Unfortunately, anti-virus companies receive up to *several hundred* new malware samples each day. Clearly, the analysis of these malware samples cannot be performed completely manually. Hence, automated solutions are necessary.

The automated methods that are available nowadays to analyze unknown executables can be divided into two broad categories. *Static analysis* is the process of analyzing a program's code without actually executing it. That is, a binary is disassembled by an automated system and the behavior of the malware sample is deducted by examining the resulting assembler code. Some static analysis methods and their drawbacks will be discussed in Section 2.1. The second category of malware analysis approaches are so called *dynamic analysis* methods. In those approaches, the binary sample is executed in a virtual machine or a simulated operating system environment. While the program is running, its interaction with the operating system (e.g., the native system calls or Windows API calls it invokes) can be recorded and later presented to an analyst or scanned for suspicious system calls. An overview of dynamic analysis approaches is given in Section 2.2.

2.1 Static Analysis Techniques

In this section, we discuss existing static code analysis techniques and point out inherent limitations that make the use of dynamic approaches appealing.

During the process of static analysis, the binary sample under analysis is usually disassembled first, which denotes the process of transforming the binary code

into corresponding assembler instructions. Once this step is completed, various control flow as well as data flow analysis techniques can be employed to draw conclusions about the functionality of the program. The advantage of static analysis is that it can cover the complete program code. Thus, if there is some malicious behavior present in the analyzed binary, it can be found by successful static analysis. However, this problem of determining the actions taken by a given code sample is undecidable in the general case so often analysis is reverted to a manual approach where the resulting assembly code is just examined by a virus analyst who then determines to the best of his knowledge if a given binary is malicious or not. Recently, a approach has been introduced that relies on so called semantic signatures[18]. This method tries to find certain behavioral patterns in assembly code defined by sequences of system calls. While this method is very promising for identifying malicious code, it only works if the disassembly of the given malware sample is successful. In Chapter3 we will show in more detail how the static analysis of malicious binaries can be impeded and how methods like[18] can thus be evaded by malicious code.

Other static binary analysis techniques [17, 61] have been introduced to detect and analyze different types of malware. However, the general problem with static analysis remains. Many interesting questions that one can ask about a program and its properties are undecidable in the general case. Of course, there exist a rich body of work on static analysis techniques that demonstrate that many problems can be approximated well in practice, often because difficult-to-handle situations occur rarely in real-world software. Unfortunately, the situation is different when dealing with malware. Because malicious code is written directly by the adversary, it can be crafted deliberately so that it is hard to analyze. In particular, the attacker can make use of binary obfuscation techniques to thwart both the disassembly and code analysis steps of static analysis approaches.

The term *obfuscation* refers to techniques that preserve the program's semantics and functionality while, at the same time, making it more difficult for the analyst to extract and comprehend the program's structures. In the context of disassembly, obfuscation refers to transformations of the binary such that the parsing of instructions becomes difficult. In [63], Linn and Debray introduced novel obfuscation techniques that exploit the fact that the Intel x86 instruction set ar-

chitecture contains variable length instructions that can start at arbitrary memory address. By inserting padding bytes at locations that cannot be reached during run-time, disassemblers can be confused to misinterpret large parts of the binary. Although their approach is limited to Intel x86 binaries, the obfuscation results against current state-of-the-art disassemblers are remarkable.

Besides obfuscation techniques to increase the difficulty of the disassembly process, the code itself can be obfuscated to make it difficult to extract the control flow of a program or to perform data flow analysis. The basic idea for such obfuscation techniques is that they can be automatically applied, but not easily undone, even if the transformation approach is known. This requirement is similar to the one that inspired the "one-way translation process" introduced in [107], or cryptography. In both cases, a process is suggested that is easy to perform in one direction, but difficult to revert.

One possibility to realize such obfuscation is the use of a primitive called *opaque constants*. Opaque constants are an extension to the idea of opaque predicates, which are defined in [21] as "boolean valued expressions whose values are known to the obfuscator but difficult to determine for an automatic deobfuscator." The difference between opaque constants and opaque predicates is that opaque constants are not boolean, but integer values. More precisely, opaque constants are mechanisms to load a constant into a processor register whose value cannot be determined statically. In Chapter 3 we will show how, based on opaque constants, a number of obfuscation transformations can be constructed that can effectively thwart static analysis of an obfuscated binary.

Finally, the code that is analyzed by a static analyzer may not necessarily be the code that is actually run. In particular, this is true for self-modifying programs that use polymorphic [101, 113] and metamorphic [101] techniques and packed executables that unpack themselves during run-time [76].

The problems of static analysis methods we addressed in this section render those approaches unsuitable for the analysis of malicious code. In the next section we will thus give an overview over dynamic binary analysis methods that, due to the nature of their techniques, are immune to the obfuscation methods presented in this section.

2.2 Dynamic Analysis Techniques

In contrast to static techniques, dynamic techniques analyze the code during runtime. While these techniques are non-exhaustive, they have the significant advantage that only those instructions are analyzed that the code actually executes. Thus, dynamic analysis is immune to obfuscation attempts and has no problems with self-modifying programs. When using dynamic analysis techniques, the question arises in which environment the sample should be executed. Of course, running malware directly on the analyst's computer, which is probably connected to the Internet, could be disastrous as the malicious code could easily escape and infect other machines. Furthermore, the use of a dedicated stand-alone machine that is reinstalled after each dynamic test run is not an efficient solution because of the overhead that is involved.

Running the executable in a virtual machine (that is, a virtualized computer) such as one provided by VMware [106] is a popular choice. In this case, the malware can only affect the virtual PC and not the real one. After performing a dynamic analysis run, the infected hard disk image is simply discarded and replaced by a clean one (i.e., so called *snapshots*). Virtualization solutions are sufficiently fast. There is almost no difference to running the executable on the real computer, and restoring a clean image is much faster than installing the operating system on a real machine. Unfortunately, a significant drawback is that the executable to be analyzed may determine that it is running in a virtualized machine and, as a result, modify its behavior. In fact, a number of different mechanisms have been published [84, 87] that explain how a program can detect if it is run inside a virtual machine. Of course, these mechanisms are also available for use by malware authors.

A PC emulator is a piece of software that emulates a personal computer (PC), including its processor, graphic card, hard disk, and other resources, with the purpose of running an unmodified operating system. It is important to differentiate emulators from virtual machines such as VMware. Like PC emulators, virtualizers can run an unmodified operating system, but they execute a statistically dominant subset of the instructions directly on the real CPU. This is in contrast to PC emulators, which simulate all instructions in software. Because all instruc-

tions are emulated in software, the system can appear exactly like a real machine to a program that is executed, yet keep complete control. Thus, it is more difficult for a program to detect that it is executed inside a PC emulator than in a virtualized environment. Note that there is one observable difference between an emulated and a real system, namely speed of execution. This fact could be exploited by malicious code that relies on timing information to detect an emulated environment.

In addition to differentiating the type of environment used for dynamic analysis, one can also distinguish and classify different types of information that can be captured during the analysis process. Many systems focus on the interaction between an application and the operating system and intercept system calls or hook Windows API calls. For example, a set of tools provided by Sysinternals [86] allows the analyst to list all running Windows processes (similar to the Windows Task Manager), or to log all Windows registry and file system activity. These tools are implemented as operating system drivers that intercept native Windows system calls. As a result, they are invisible to the application that is being analyzed. They cannot, however, intercept and analyze Windows API calls or other user functions. On the other hand, tools [52] exist that can intercept arbitrary user functions, including all Windows API calls. This is typically realized by rewriting target function images. The original function is preserved as a subroutine and callable through a trampoline. Unfortunately, the fact that code needs to be modified can be detected by malicious code that implements integrity checking.

Most current dynamic analysis systems have complete control over the sample program running in a virtual PC environment. They can usually intercept and analyze both native Windows operating system calls as well as Windows API calls while being invisible to malicious code. The complete control offered by a PC emulator potentially allows the analysis that is performed to be even more fine-grained. Similar to the functionality typically provided by a debugger, the code under analysis can be stopped at any point during its execution and the process state (i.e., registers and virtual address space) can be examined. Unlike a debugger, however, those systems do not have to resort to breakpoints, which are known to cause problems when used for malicious code analysis [104]. The reason is that software breakpoints directly modify the executable and thus, can be detected

by code integrity checks. Also, malicious code was found in the wild that used processor debug registers for its computations, thereby breaking hardware breakpoints.

A number of dynamic analysis systems exist that use one of the approaches described above [2, 75, 110]. Those systems allow users to upload malicious code samples and output detailed reports on the behavior of the code sample. Unfortunately, only one code path of the binary under analysis can be examined in one run by conventional dynamic analysis systems. Thus, if for example the malicious code sample is able to determine that it is being analyzed, it can just exit prematurely. In that case, the generated reports will not show any malicious behavior at all. To mitigate those problems we will present a novel method to increase the code coverage of dynamic analysis in Chapter 4.

Chapter 3

Limits of Static Analysis for Malware Detection

Current systems to detect malicious code (most prominently, virus scanners) are largely based on syntactic signatures. That is, these systems are equipped with a database of regular expressions that specify byte or instruction sequences that are considered malicious. A program is declared malware when one of the signatures is identified in the program's code. However, syntactic signatures have two drawbacks. First, specifying precise, syntactic signatures makes it necessary to update the signature database whenever a previously unknown malware sample is found. Hence, there is always a window of vulnerability between the appearance of a new malicious code instance and the availability of a signature that can detect it. Second, malicious code can make use of simple program transformation (or obfuscation) techniques. Using such techniques, the syntactic layout of the code is modified, thus, evading detection.

Polymorphism and *metamorphism* are two obfuscation techniques that are commonly employed by malware authors. In the case of polymorphism, the actual malware body is encrypted and prepended by a short decryption routine. Whenever the malicious code is executed, the decryption routine first decrypts the malicious code and then executes it. Of course, a different encryption key can be used for each malware instance, making it impossble to specify a signature for the encrypted body itself. From the point of view of the malware author, the prob-

lem of evading detection has only shifted though, as virus scanners now target the decryption routines.

Metamorphic techniques, on the other hand, do not encrypt the code but attempt to directly change its layout so that signatures no longer match. For example, the use of one register can be replaced by the use of another one that is not active in the current code sequence (a technique called *register reassignment*). Other techniques achieve changes in the code layout by shuffling independent instructions (*code transposition*) or by inserting additional instructions with no effect on the program's behavior (*dead code insertion*). Finally, an instruction sequence can be substituted by another one with the same semantics (*instruction substitution*), a mechanism favored by complex instruction set architectures such as the Intel `x86`.

Obviously, both polymorphic and metamorphic techniques can be combined. It is a common approach that malicious code is first encrypted before metamorphic techniques are applied to obfuscate the decryption routine.

Recent work [17] has demonstrated that techniques such as *polymorphism* and *metamorphism* are successful in evading commercial virus scanners. The reason is that syntactic signatures are ignorant of the semantics of instructions. To address this problem, a novel class of *semantics-aware* malware detectors was proposed. These detectors [18, 55, 61] operate with abstract models, or templates, that describe the behavior of malicious code. Because the syntactic properties of code are (largely) ignored, these techniques are (mostly) resilient against the evasion attempts discussed above. The premise of semantics-aware malware detectors is that semantic properties are more difficult to morph in an automated fashion than syntactic properties. While this is most likely true, the extent to which this is more difficult is less obvious. On one hand, semantics-aware detection faces the challenge that the problem of deciding whether a certain piece of code exhibits a certain behavior is undecidable in the general case. On the other hand, it is also not trivial for an attacker to automatically generate semantically equivalent code.

The question that we address in this thesis is the following: *How difficult is it for an attacker to evade semantics-based malware detectors that use powerful static analysis to identify malicious code?* We try to answer this question by introducing a binary code obfuscation technique that makes it difficult for an advanced,

semantics-based malware detector to properly determine the effect of a piece of code. For this obfuscation process, we use a primitive known as *opaque constant*, which denotes a code sequence to load a constant into a processor register whose value cannot be determined statically. Based on opaque constants, we build a number of obfuscation transformations that are difficult to analyze statically.

Given our obfuscation scheme, the next question that needs to be addressed is how these transformations should be applied to a program. The easiest way, and the approach chosen by most previous obfuscation approaches [21, 108], is to work on the program's source code. Applying obfuscation at the source code level is the normal choice when the distributor of a binary controls the source (e.g., to protect intellectual property). For malware that is spreading in the wild, source code is typically not available. Also, malware authors are often reluctant to revealing their source code to make analysis more difficult. Thus, to guard against objections that our presented threats are unrealistic, we present a solution that operates directly on binaries.

The core contributions we present in this chapter are as follows:

- We present a binary obfuscation scheme based on the idea of opaque constants. This scheme allows us to demonstrate that static analysis of advanced malware detectors can be thwarted by scrambling control flow and hiding data locations and usage.

- We introduce a binary rewriting tool that allows us to obfuscate Windows and Linux binary programs for which no source code or debug information is available.

- We present experimental results that demonstrate that semantics-aware malware detectors can be evaded successfully. In addition, we show that our binary transformations are robust, allowing us to run real-world obfuscated binaries under both Linux and Windows.

The code obfuscation scheme introduced in this thesis provides a strong indication that static analysis alone might not be sufficient to detect malicious code. In particular, we introduce an obfuscation scheme that is provably hard to analyze

statically. Because of the many ways in which code can be obfuscated and the fundamental limits in what can be decided statically, we firmly believe that dynamic analysis is a necessary complement to static detection techniques. The reason is that dynamic techniques can monitor the instructions that are actually executed by a program and, thus, are immune to many code obfuscating transformations.

3.1 Code Obfuscation

In this section, we present the concepts of the obfuscation transformations that we apply to make the code of a binary difficult to analyze statically. As with most obfuscation approaches, the basic idea behind our transformations is that either some instructions of the original code are replaced by program fragments that are semantically equivalent but more difficult to analyze, or that additional instructions are added to the program that do not change its behavior. Many of our transformations rely on the existence of an obfuscation primitive that we call *opaque constant*, which we discuss in the next section. The following sections then introduce obfuscation techniques that can be built on top of this obfuscation primitive.

3.1.1 Opaque Constants

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

Simple Opaque Constant Calculation Figure 3.1 shows one approach to create a code sequence that makes use of random input and different intermediate

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

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

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

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

Figure 3.1: Opaque constant calculation

variable values on different branches. In this code sequence, the value `unknown` is a random value loaded during runtime. To prepare the opaque constant calculation, the bits of the constant that we aim to create have to be randomly partitioned into two groups. The values of the arrays `zero` and `one` are crafted such that after the `for` loop, all bits of the first group have the correct, final value, while those of the second group depend on the random input (and thus, are unknown). Then, using the appropriate values for `set_ones` and `set_zeros`, all bits of the second group are forced to their correct values (while those of the first group are left unchanged). The result is that all bits of `constant` hold the desired value at the end of the execution of the code.

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

values doubles after each loop iteration. In such a case, the static analyzer would likely have to resort to approximation, in which case the exact knowledge of the constant is lost.

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

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

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

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

where $V_{ij} \in \{v_1, ..., v_m\}$ and $v_1, ..., v_m$ are Boolean variables whose value can be either *true* or *false*. The task is now to determine if there exists an assignment for

the variables v_k such that the given formula is satisfied (i.e., the formula evaluates to true). 3SAT has been proven to be NP-complete in [54].

```
boolean v1, ..., vm, v̄1, ..., v̄m;

boolean *V11, *V12, *V13;
...
boolean *Vn1, *Vn2, *Vn3;

constant = 1;
for (i = 0; i < n; ++i)
  if !(*Vi1) && !(*Vi2) && !(*Vi3)
    constant = 0;
```

Figure 3.2: Opaque constant based on 3SAT

Consider the code sequence in Figure 3.2. In this primitive, we define m boolean variables $v_1 \dots v_m$, which correspond directly to the variables in the given 3SAT formula. By $\overline{v_1} \dots \overline{v_m}$, we denote their negations. The pointers V_{11} to V_{n3} refer to the variables used in the various clauses of the formula. In other words, the pointers V_{11} to V_{n3} encode a 3SAT problem based on the variables $v_1 \dots v_m$. The loop simply evaluates the encoded 3SAT formula on the input. If the assignment of variables $v_1 \dots v_m$ does not satisfy the formula, there will always be at least one clause i that evaluates to false. When the check in the loop is evaluated for that specific clause, the result will always be true (as the check is performed against the negate of the clause). Therefore, the opaque constant will be set to 0. On the other hand, if the assignment satisfies the encoded formula, the check performed in the loop will never be true. Therefore, the value of the opaque constant is not overwritten and remains 1.

In the opaque constant presented in Figure 3.2, the 3SAT problem (that is, the pointers V_{11} to V_{n3}) is prepared by the obfuscator. However, the actual assignment of boolean values to the variables $v_1 \dots v_m$ is randomly performed during runtime. Therefore, the analyzer cannot immediately evaluate the formula. The trick of our opaque constant is that the 3SAT problem is prepared such that the

formula is not satisfiable. Thus, independent of the actual input, the constant will always evaluate to 0. Of course, when a constant value of 1 should be generated, we can simply invert the result of the satisfiability test. Note that it is possible to efficiently generate 3SAT instances that are not satisfiable with a high probability [91]. A static analyzer that aims to exactly determine the possible values of our opaque constant has to solve the instance of the 3SAT problem. Thus, 3SAT is reducible in polynomial time to the problem of exact static analysis of the value of the given opaque constant.

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

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

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

To analyze a program that is obfuscated with basic block chaining, the analyzer cannot rely on the fact that the encoded formula is always unsatisfiable. Also, when randomly executing a few sample inputs, it is unlikely that the analyzer chooses values that solve a satisfiable formula. The only way to dissect an opaque constant would be to first identify the basic block(s) that precede a certain formula and then determine whether the input values stored in this block

satisfy the 3SAT problem. However, finding these blocks is not trivial, as the control flow of the program is obfuscated to make this task difficult (see the following Section 3.1.2 for more details). Thus, the analysis would have to start at the program entry point and either execute the program dynamically or resort to an approach similar to whole program simulation in which different branches are followed from the start, resolving opaque constants as the analysis progresses. Obviously, our obfuscation techniques fail against such methods, and indeed, this is consistent with an important point that we intend to make in this chapter: dynamic analysis techniques are a promising and powerful approach to deal with obfuscated binaries.

3.1.2 Obfuscating Transformations

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

Control Flow Obfuscation

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

The idea to obfuscate the control flow is to replace unconditional jump and call instructions with a sequence of instructions that do not alter the control flow, but make it difficult to determine the target of control transfer instructions. In

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

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

Data Location Obfuscation

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

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

an opaque constant primitive. Then, the now unknown value (from the point of view of the static analyzer) is used as offset to the base or stack pointer.

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

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

Data Usage Obfuscation

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

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

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

3.2 Binary Transformation

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

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

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

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

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

Our rewriting tool targets both the Linux ELF [102] and the Windows PE [65] file formats. In general, Linux applications are significantly easier to disassemble than Windows binaries. A reason for this is that almost all ELF binaries are generated by the GNU compiler collection (GCC). The usage of the same compiler back-end for nearly all files results in a consistent file landscape. Also, the

code produced by GCC is well-structured. PE files, on the other hand, are produced by various popular commercial compilers (e.g., MS Visual Studio, Borland, Metrowerks CodeWarrior) that generate code using different conventions and layouts. Furthermore, Windows compilers do not separate code and data regions as cleanly as GCC. In practice, this means that data is almost always inserted before the first and after the last valid instruction in the code section of PE files. Sometimes, compilers also insert user data directly into the code segment.

Using the recursive disassembler approach and our heuristics, our binary rewriting tool is able to correctly obfuscate many (although not all) real-world binaries. More detailed results on the robustness of the tool are provided in Section 3.3.

3.3 Evaluation

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

3.3.1 Evasion Capabilities

To demonstrate that the presented obfuscation methods can be used to effectively change the structure of a binary so that static analysis tools fail to recognize the obfuscated code, we conducted tests with real-world malware. We used our tool to morph three worm programs and then analyzed the obfuscated binaries using an advanced static analysis tool [55] as well as four popular commercial virus scanners.

The malware samples that we selected for our experiments were the `A` and `F` variants of the `MyDoom` worm and the `A` variant of the `Klez` worm. We chose these samples because they were used in the evaluation of the advanced static analysis tool in [55]. Thus, the tool was equipped with appropriate malware specifications to detect these worms. In order to obfuscate the malicious executables,

we deployed the evasion techniques introduced in Section 3.1 using both the simple opaque constants and the one based on the 3SAT problem.

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

Before the experiment, we verified that all scanners correctly identified the worms. Then, we obfuscated the three malicious code samples, ensured that the malware was still operating correctly, and ran the virus scanners on them. The results are shown in Table 3.1. In this table, an "X" indicates that the scanner was no longer able to detect the malware.

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

Table 3.1: Evasion results for four commercial virus scanners

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

was successfully evaded. Indeed, it is relatively easy for malicious code to encrypt the data section using a different key for each instance. Hence, data signatures are not too difficult to evade.

Advanced Malware Detection (Model Checking): Because it is widely known that existing commercial virus scanners typically employ pattern-based signatures, the ability to evade their detection is not too surprising. In order to verify the efficiency of our obfuscation techniques on a more advanced malware detector, we obtained the system presented in [55] from its authors. This detector first creates a disassembly of the binary under analysis by using IDA Pro [35]. Then, model checking is used to search for the existence of a generic code template that characterizes malicious behavior. In particular, the tool attempts to identify code sequences in the program that copy this program's binary to another location in the file system. More precisely, a malicious code sequence is defined as a call to the `GetModuleFileNameA` Windows API function, followed by an invocation of the `CopyFileA` function. The exact specification as presented in [55] is shown below.

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

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

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

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

Semantics-Aware Malware Detection: Another system that uses code templates instead of patterns to specify malicious code was presented in [18]. We did a theoretical examination of the evasion capabilities of the approach presented here against this system and the first problem we found is clearly its dependency on a correct disassembly of the binary. The system uses IDA Pro [35] to generate such disassembly outputs but this disassembler produces incorrect results when confronted with control flow obfuscation. A second problem is the dependency of some code templates (or semantic signatures) on the fact that certain constants must be recognized as equivalent. Consider the template that specifies a decryption loop, which describes the behavior of programs that unpack or decrypt themselves to memory. According to [18], such a template consists of "(1) a loop that processes data from a source memory area and writes data to a destination

memory area, and (2) a jump that targets the destination area." Clearly, the detector must be able to establish a relationship between the memory area where the code is written to and the target of the jump. However, when using data location obfuscation, the detector cannot statically determine where data is written to, and by using obfuscated jumps, it also cannot link this memory area with the target of the control flow instruction. Finally, semantic signatures can make use of define-use chains to link the location where a variable is set and the location where it is used. By using data usage obfuscation, however, such define-use chains can be broken. Thus, we believe that a malware binary obfuscated by the methods presented in this thesis will not be deemed malicious anymore by the analysis system presented in [18].

3.3.2 Transformation Robustness

In this section, we discuss the robustness of the applied modifications (i.e., does a program run without problems after it is obfuscated) as well as their size and performance impact. When testing whether obfuscation was successful, one faces the problem of test coverage. That is, it is not trivial to demonstrate that the obfuscated program behaves exactly like the original one. Because we operate directly on binaries, our biggest challenge is the correct distinction between code and data regions. When the disassembly step confuses code and data, addresses are updated incorrectly and the program crashes. We observed that disassembler errors quickly propagate through the program. Thus, whenever the binary rewriting fails, the obfuscated programs typically crash quickly. On the other hand, once an obfuscated application was running, we observed few problems during the more extensive tests we conducted. Thus, the mere fact that a program can be launched provides a good indication for the success of the transformation process. Of course, this is no guarantee for the correctness of the obfuscation process in general.

Linux Binaries. In general, rewriting ELF binaries for Linux works very well. Our first experiment was performed on the GNU coreutils. This software package consists of 93 applications that can be found on virtually every Linux machine.

Part of the coreutils package is a test script that performs 210 checks on various applications. To assess the robustness of our transformations, we rewrote all 93 applications using all obfuscation transformations introduced previously. We then ran the test script again, and all 210 checks were passed without problems.

As a second experiment, we obfuscated all applications in the `/usr/bin/` directory on a machine running Ubuntu Linux 5.10. For this test, we rewrote 774 applications. When manually checking these applications, we recorded eleven programs that crashed with a segmentation fault. Among these programs were large, complex applications such as Emacs and Evolution or the linker. Of those programs that were successfully rewritten, we extensively used and tested applications such as the instant messenger gaim (806 KB), vim (1,074 KB), xmms (991 KB) and the Opera web browser (12,059 KB). These applications exhibited no problems during regular use, which underlines the robustness of the transformations even for large binaries. Given that we operate directly on binaries without any available program information and considering the fact that code rewriting was successful for many applications, including GCC and Opera, we can conclude that our binary transformation process is quite robust.

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

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

3.3.3 Size and Performance

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

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

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

3.3.4 Possible Countermeasures

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

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

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

3.3.5 Summary

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

To be able to assess the effectiveness of such an obfuscation approach, we developed a binary rewriting tool that allows us to perform the necessary mod-

ifications. This tool is capable of successfully morphing a significant fraction of Linux and Windows binaries, even when no additional information (e.g., relocation data) is present. Using the tool, we obfuscated well-known worms and demonstrated that neither commercial virus scanners nor a more advanced static analysis tool based on model checking could identify the transformed programs.

While it is conceivable to improve static analysis to handle more advanced obfuscation techniques, there is a fundamental limit in what can be decided statically. In particular, we presented a construct based on the 3SAT problem that is provably hard to analyze. Thus, even if limits of static analysis are of less concern when attempting to find bugs in benign programs, they are problematic and worrisome when analyzing malicious, binary code that is deliberately designed to resist analysis.

In this chapter, we demonstrated that static techniques alone are not sufficient to identify malicious code. This is the motivation to concentrate this thesis on systems that apply dynamic analysis methods. In Chapter 4 we will present an approach to extend dynamic malware analysis systems that are immune to the obfuscation techniques presented in this chapter. Those systems are better suited to analyze current malware samples that often already apply polymorphism or metamorphism than the static analysis methods we briefly discussed.

Chapter 4

Exploring Multiple Execution Paths for Malware Analysis

Nowadays, a number of systems are in use to dynamically analyze unknown binaries. These systems, such as CWSandbox [110], the Norman SandBox [75], Cobra [105], or Anubis [2], automatically load the sample to be analyzed into a virtual machine environment and execute it. While the program is running, its interaction with the operating system is recorded. Typically, this involves recording which system calls are invoked, together with their parameters. The result of an automated analysis is a report that shows what operating system resources (e.g., files or Windows registry entries) a program has created or accessed. Some tools also allow the system to connect to a local network (or even the Internet) and monitor the network traffic. Usually, the generated reports provide human analysts with an overview on the behavior of the sample and allow them to quickly decide whether a closer, manual analysis is required. Hence, these automated systems free the analysts of the need to waste time on already known malware. Also, some tools are already deployed on the Internet and act as live analysis back-ends for honeypot installations such as Nepenthes [6]. Unfortunately, current analysis systems also suffer from a significant drawback: their analysis is based on a *single* execution trace only. That is, their reports only contain the interaction that was observed when the sample was run in a particular test environment at a certain

point in time. Unfortunately, this approach has the potential to miss a significant fraction of the behavior that a program might exhibit under varying circumstances.

Malware programs frequently contain checks that determine whether certain files or directories exist on a machine and only run parts of their code when they do. Others require that a connection to the Internet is established or that a specific mutex object does not exist. In case these conditions are not met, the malware may terminate immediately. This is similar to malicious code that checks for indications of a virtual machine environment, modifying its behavior if such indications are present in order to make its analysis in a virtual environment more difficult. Other functionality that is not invoked on every run are malware routines that are only executed at or until a certain date or time of day. For example, some variants of the Bagle worm included a check that would deactivate the worm completely after a certain date. Another example is the Michelangelo virus, which remains dormant most of the time, delivering its payload only on March 6 (which is Michelangelo's birthday). Of course, functionality can also be triggered by other conditions, such as the name of the user or the IP address of the local network interface. Finally, some malware listens for certain commands that must be sent over a control channel before an activity is started. For example, bots that automatically log into IRC servers often monitor the channel for a list of keywords that trigger certain payload routines.

When the behavior of a program is determined from a single run, it is possible that many of the previously mentioned actions cannot be observed. This might lead a human analyst to draw incorrect conclusions about the risk of a certain sample. Even worse, when the code fails at an early check and immediately exits, the generated report might not show any malicious activity at all. One possibility to address this problem is to attempt to increase test coverage. This could be done by running the executable in different environments, maybe using a variety of operating system versions, installed applications, and data/time settings. Unfortunately, even with the help of virtual machines, creating and maintaining such a testing system can be costly. Also, performing hundreds of tests with each sample is not very efficient, especially because many environmental changes have no influence on the program execution. Moreover, in cases where malicious code is expecting certain commands as input or checking for the existence of non-standard files

(e.g., files that a previous exploit might have created), it is virtually impossible to trigger certain actions.

In this chapter, we propose a solution that addresses the problem of test coverage and that allows automated malware analysis systems to generate more comprehensive reports. The basic idea is that we explore multiple execution paths of a program under test, but the exploration of different paths is driven by monitoring how the code uses certain inputs. More precisely, we dynamically track certain input values that the program reads (such as the current time from the operating system, the content of a file, or the result of a check for Internet connectivity) and identify points in the execution where this input is used to make control flow decisions. When such a decision point is identified, we first create a snapshot of the current state of the program execution. Then, the program is allowed to continue along one of the execution branches, depending on the actual input value. Later, we return to the snapshot and rewrite the input value such that the other branch is taken. This allows us to explore both program branches. In addition, we can determine under which conditions certain code paths are executed.

For a simple example, consider a program that checks for the presence of a file. During execution, we track the result of the operating system call that checks for the existence of that file. When this result is later used in a conditional branch by the program, we store a snapshot of the current execution state. Suppose, for example, that the file does not exist, and the program quickly exits. At this point, we rewind the process to the previously stored state and rewrite the result such that it does reports the file's existence. Then, we can explore the actions that the program performs under the condition that the file is there.

We have developed a system for Microsoft Windows that allows us to dynamically execute programs and track the input that they read. Also, we have implemented a mechanism to take snapshots of executing processes and later revert to previously stored states. This provides us with the means to explore the execution space of malware programs and to observe behavior that is not seen by traditional malware analysis environments. To demonstrate the feasibility of our approach, we analyzed a large number of real-world malware samples. In our experiments, we were able to identify time checks that guarded damage routines and different behavior depending on existence of certain files. Also, we were able to automat-

ically extract a number of command strings for a bot with their corresponding actions.

To summarize, the contributions of this chapter are as follows:

- We propose a dynamic analysis technique that allows us to create comprehensive reports on the behavior of malicious code. To this end, our system explores multiple program paths, driven by the input that the program processes. Also, our system reports the set of conditions on the input under which particular actions are triggered.
- We developed a tool that analyzes Microsoft Windows programs by executing them in a virtual-machine-based environment. Our system keeps track of user input and can create snapshots of the current process at control flow decision points. In addition, we can reset a running process to a previously stored state and consistently modify its memory such that the alternative execution path is explored.
- We evaluated our system on a large number of real-world malware samples and demonstrate that we were able to identify behavior that cannot be observed in single execution traces.

4.1 System Overview

The techniques described in this chapter are an extension to a system for automated malware analysis [9]. This tool is based on Qemu [10], a fast virtual machine emulator. Using Qemu's emulation of an Intel x86 host system, a Windows 2000 guest operating system is installed. The choice of Windows and the Intel x86 architecture was motivated by the fact that the predominant fraction of malware is developed for this platform.

The existing analysis tool implements some virtual machine introspection capabilities; in particular, it is able to attribute each instruction that is executed by the emulated processor to an operating system process (or the kernel) of the guest system. This allows us to track only those system calls that are invoked by the code under analysis. Also, the system provides a mechanism to copy the content

of complex data structures, which can contain pointers to other objects in the process' virtual address space, from the Windows guest system into the host system. This is convenient in order to be able to copy the system call arguments from the emulated system into the analysis environment. Unfortunately, the existing system only collected a single execution trace.

Multiple execution paths. To address the problem that a single execution trace typically produces only part of the complete program behavior, we extended the analysis tool with the capability to explore multiple execution paths. The goal is to obtain a number of different execution paths, and each path possibly reveals some specific behavior that cannot be observed in the other traces. The selection of *branching points* – that is, points in the program execution where both alternative continuations are of interest – is based on the way the program processes input data. More precisely, when a control flow decision is based on some input value that was previously read via a system call, the program takes one branch (which depends on the outcome of the concrete check). At this point, we ask ourselves the following question: Which behavior could be observed if the input was such that the other branch was taken?

To answer this question, we label certain inputs of interest to the program and dynamically track their propagation during execution. Similar to the propagation of taint information used by other authors in previous work [28, 73], our system monitors the way these input values are moved and manipulated by the process. Whenever we detect a control flow decision based on a labeled value, the current content of the process address space is stored. Then, execution continues normally. When the process later wishes to terminate, it is automatically reset to the previously stored snapshot. This is done by replacing the current content of the process address space with the previously stored values. In addition, we rewrite the input value that was used in the control flow decision such that the outcome of this decision is reversed. Then, the process continues its execution along the other branch. Of course, it is possible that multiple branchings in a row are encountered. In this case, the execution space is explored by selecting continuation points in a depth-first order.

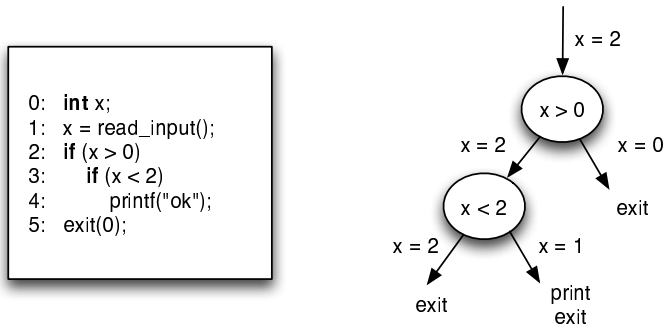


Figure 4.1: Exploration of multiple execution paths.

For an example on how multiple execution paths of a program can be explored, consider Figure 4.1. Note that although this example is shown in C code (to make it easier to follow), our system works directly on x86 binaries. When the program is executed, it first receives some input and stores it into variable x (on Line 1). Note that because x is considered interesting, it is labeled. Assume that in this concrete run, the value stored into x is 2. On Line 2, x is compared to 0. At this point, our system detects a comparison operation that involves labeled data. Thus, a snapshot of the current process is created. Then, the process is allowed to continue. Because the condition is satisfied, the if-branch is taken and we record the fact that x has to be larger than 0. On Line 3, the next check fails. However, because the comparison again involves labeled data, another snapshot is created. This time, the process continues on the else-branch and is about to call `exit`. Because there are still unexplored paths (i.e., there exist two states that have not been visited), the process is reverted to the previous (second) state. Our system inspects the comparison at Line 3 and attempts to rewrite x such that the check succeeds. For this, the additional constraint $x > 0$ has to be observed. This yields a solution for x that equals 1. The value of x is updated to 1 and the process is restarted. This time, the `print` statement on Line 4 is invoked. When the process is about to exit on Line 5, it is reset to the first snapshot. This time, the system searches a value for x that fails the check on Line 2. Because there are no additional constraints for x, an arbitrary, non-positive integer is selected and the process continues along the else-branch. This time, the call to `exit` is

permitted, and the analysis process terminates with a report that indicates that a call to `print` was found under the condition that the input x was 1 (but not 0 or 2).

Consistent memory updates. Unfortunately, when rewriting a certain input value to explore an alternative execution path, it is typically not sufficient to change the single memory location that is used by the control flow decision. Instead, it is necessary to consistently update (or rewrite) all values in the process address space that are related to the input. The reason is that the original input value might have been copied to other memory locations, and even used by the program as part of some previous calculations. When only a single instance of the input is modified, it is possible that copies of the original value remain in the program's data section. This can lead to the execution of invalid operations or the exploration of impossible paths. Thus, whenever an input value is rewritten, it is necessary to keep the program state consistent and appropriately update all copies of the input, as well as results of previous operations that involve this value. Also, we might not have complete freedom when choosing an alternative value for a certain input. For example, an input might have been used in previous comparison operations and the resulting constraints need to be observed when selecting a value that can revert the control flow decision at a branching point. It is even possible that no valid alternative value exists that can lead to the exploration of the alternative path. Thus, to be able to consistently update and input and its related values, it is necessary to keep track of *which* memory locations depend on a certain input and *how* they depend on this value.

4.2 Path Exploration

To be able to explore multiple program paths, two main components are required. First, we need a mechanism to decide when our system should analyze both program paths. To this end, we track how the program uses data from certain input sources. Second, when an interesting branching point is located, we require a mechanism to save the current program state and reload it later to explore the al-

ternative path. The following two subsections discuss these two components in more detail.

4.2.1 Tracking Input

In traditional taint-based systems, it is sufficient to know that a certain memory location depends on one or more input values. To obtain this information, such systems typically rely on three components: a set of taint sources, a shadow memory, and extensions to the machine instructions that propagate the taint information.

Taint sources are used to initially assign labels to certain memory locations of interest. For example, Vigilante [27] is a taint-based system that can detect computer worms that propagate over the network. In this system, the network is considered a taint source. As a result, each new input byte that is read from the network card by the operating system receives a new label. The shadow memory is required to keep track of which labels are assigned to which memory locations at a certain point in time. Usually, a shadow byte is used for each byte of physical machine memory. This shadow byte stores the label(s) currently attached to the physical memory location. Finally, extensions to the machine instructions are required to propagate taint information when an operation manipulates or moves labeled data. The most common propagation policy ensures that the result of an operation receives the union of the labels of the operation's arguments. For example, consider an `add` machine instruction that adds the constant value 10 to a memory location M_1 and stores the result at location M_2. In this case, the system would use the shadow memory to look up the label attached to M_1 and attach this label to M_2. Thus, after the operation, both locations M_1 and M_2 share the same label (although their content is different).

In principle, we rely on a taint-based system as previously described to track how the program under analysis processes input values. That is, we have a number of taint sources that assign labels to input that is read by the program, and we use a shadow memory to keep track of the current label assigned to each memory location (including the processor registers). Taint sources in our system are mostly system calls that return information that we consider relevant for the behavior of malicious code. This includes system calls that access the file system (e.g., check

for existence of file, read file content), the Windows registry, and the network. Also, system calls that return the current time or the status of the network connection are interesting. Whenever a relevant function (or system call) is invoked by our program, our system automatically assigns a new label to each memory location that receives this function's result. Sometimes, this means that a single integer is labeled. In other cases, for example, when the program reads from a file or the network, the complete return buffer is labeled, using one unique label per byte.

Inverse mapping. In addition to the shadow memory, which maps memory locations to labels, we also require an *inverse mapping*. The inverse mapping stores, for each label, the addresses of all memory locations that currently hold this label. This information is needed when a process is reset to a previously stored state and a certain input variable must be rewritten. The reason is that when a memory location with a certain label is modified, it is necessary to simultaneously change all other locations that have the same label. Otherwise, the state of the process becomes inconsistent. For example, consider the case in which the value of labeled input x is copied several times before it is eventually stored at memory location y. Furthermore, assume that y is used as argument by a conditional branch. To explore the alternate execution branch, the content of y must be changed. However, via a chain of intermediate locations, this value ultimately depends on x. Thus, all intermediate locations need to be modified appropriately. To this end, a mapping is required that helps us to quickly identify all locations that currently share the same label.

To underline the importance of a consistent memory update, consider the example in Figure 4.2. Assume that the function `read_input` on Line 1 is a taint source. Thus, when the program executes this function, variable *x* is labeled. In our example, the program initially reads the value 0. When the `check` routine is invoked, the value of variable *x* is copied into the parameter *magic*. As part of this assignment, the variable *magic* receives the label of *x*. When *magic* is later used in the check on Line 7, a snapshot of the current state is taken (because the outcome of a conditional branch depends on a labeled value). Execution continues but quickly terminates on Line 8. At this point, the process is reverted to the

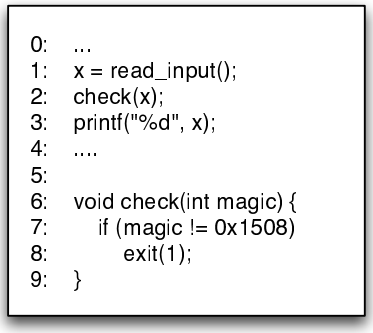


Figure 4.2: Consistent memory updates.

previously stored snapshot and our system determines that the value of *magic* has to be rewritten to $0x1508$ to take the if-branch. At this point, the new value has to be propagated to all other locations that share the same label (in our case, the variable *x*). Otherwise, the program would incorrectly print the value of 0 instead of $0x1508$ on Line 3.

Linear dependencies. In the previous discussion, the initial input value was copied to new memory locations before being used as an argument in a control flow decision. In that case, rewriting this argument implied that all locations that share the same label had to be updated with the same value. So far, however, we have not considered the case when the initial input is not simply copied, but used as operand in calculations. Using the straightforward taint propagation mechanism outlined above, the result of an operation with a labeled argument receives this argument's label. This also happens when the result of an operation has a different value than the argument. Unfortunately, this leads to problems when rewriting a variable at a snapshot point. In particular, when different memory locations share the same label but hold different values, one cannot simply overwrite these memory locations with a single, new value.

We solve this problem by assigning a *new label* to the result of any operation (different than copying) that involves labeled arguments. In addition, we have to record how the value with the new label depends on the value(s) with the old label(s). This is achieved by creating a new constraint that captures the relationship

between the old and new labels, depending on the semantics of the operation. The constraint is then added to a *constraint system* that is maintained as part of the execution state of the process. Consider the simple example where a value with label l_0 is used by an `add` operation that increases this value by the constant 10. In this case, the result of the operation receives a new label l_1. In addition, we record the fact that the result of the operation with l_1 is equal to the value labeled by l_0 plus 10. That is, the constraint $l_1 = l_0 + 10$ is inserted into the constraint system. The approach works similarly when two labeled inputs, one with label l_0 and the other with label l_1 are summed up. In this case, the result receives a new label l_2 and we add the constraint $l_2 = l_0 + l_1$.

In our current system, we can only model linear relationships between input variables. That is, our constraint system is a linear constraint system that can store terms in the form of $\{c_n * l_n + c_{n-1} * l_{n-1} + \ldots + c_1 * l_1 + c_0\}$ where the c_i are constants. These terms can be connected by equality or inequality operators. To track linear dependencies between labels, the taint propagation mechanism of the machine instructions responsible for addition, subtraction, and multiplication had to be extended.

Using the information provided by the linear constraint system, it is possible to correctly update all memory locations that depend on an input value x via linear relationships. Consider the case where a conditional control flow decision uses a value with label l_n. To explore the alternative branch of this decision, we have to rewrite the labeled value such that the outcome of the condition is reverted. To do this consistently, we first use the linear constraint system to identify all labels that are related to l_n. This provides us with the information which memory locations have some connection with l_n, and thus, must be updated as well. In a second step, a linear constraint solver is used to determine concrete values for these memory locations.

Two labels l_s and l_t are related either (a) when they appear together in the same constraint or (b) when there exists a sequence of labels $\{l_{i_0}, \ldots, l_{i_n}\}$ such that $l_s = l_{i_0}$, $l_t = l_{i_n}$, and $l_i, l_{i+1}\ \forall_{i=0}^{n-1}$ appear in the same constraint. More formally, the binary relation *related* is the transitive closure of the binary relation *appears in the same constraint*. Thus, when a value with label l_n should be rewritten, we first determine all labels that are *related* to l_n in the constraint system. Then, we

extract all constraints that contain at least one of the labels related to l_n. This set of constraints is then solved, using a linear constraint solver (we use the Parma Polyhedral Library [7]).

When the constraint system has no solution, the labeled value cannot be changed such that the outcome of the condition is reverted. In this case, our system cannot explore the alternative path, and it continues with the next snapshot stored. When a solution is found, on the other hand, this solution can be directly used to consistently update the process' state. To this end, we can directly use, for each label, the value that the solver has determined to update the corresponding memory locations. This works because all (linear) dependencies between values are encoded by the respective constraints in the constraint system. That is, a solution of the constraint system respects the relationships that have to hold between memory locations. All memory locations that share the same label receive the same value. However, as expected, when memory locations have different labels, they can also receive different values. These values respect the relationships introduced by the operation previously executed by the process and captured by the corresponding constraints in the constraint system.

To illustrate the concept of linear dependencies between values and to show how their dependencies are captured by the constraint system, consider Figure 4.3. The example shows the labels and constraints that are introduced when a simple `atoi` function is executed. The goal of this function is to convert a string into the integer value that this string represents. For this example, we assume that the function is executed on a string *str* with three characters; the first two are the ASCII character equivalent of the number 0 (which is 30). The third one has the value 0 and terminates the string. We assume that interesting input was read into the string; as a result, the first character *str[0]* has label l_0 and *str[1]* has label l_1.

The figure shows the initial mapping between program variables and labels. For this initial state, no constraints have been identified yet. After the first loop iteration, it can be seen that the variables *c* and *sum* are also labeled. This results from the operations on Line 7 and Line 8, respectively. The relationship between the variables are captured by the two constraints. Because *sum* was 0 before this loop iteration, variables *sum* and *c* hold the same value. This is expressed by the constraint $l_3 = l_2$. Note that this example is slightly simplified. The reason is

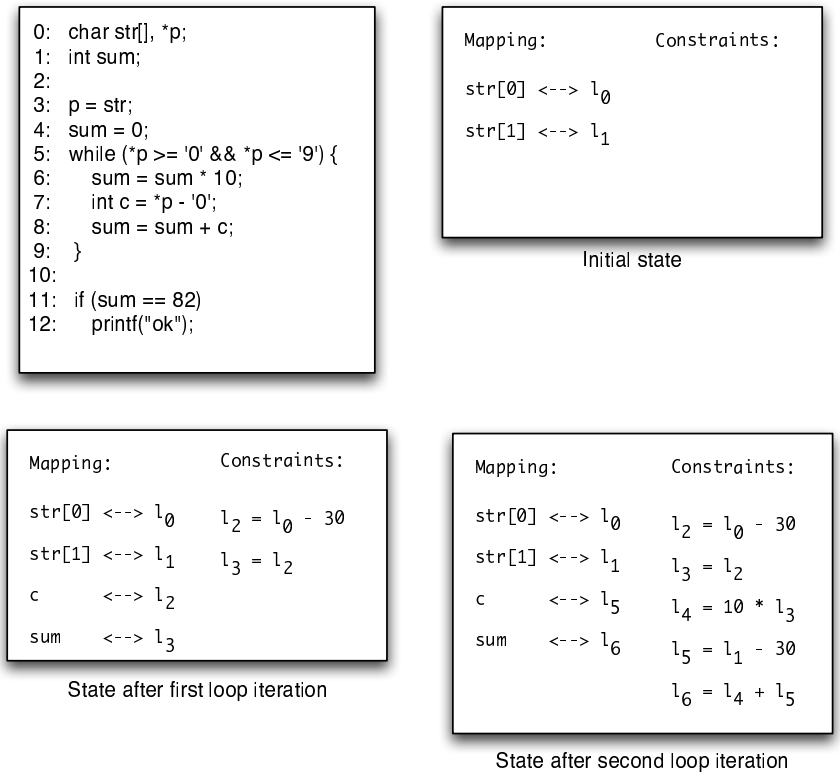


Figure 4.3: Constraints generated during program execution.

that the checks performed by the while-statement on Line 5 lead to the creation of additional constraints that ensure that the values of *str[0]* and *str[1]* are between 30 (ASCII value for '0') and 39 (ASCII value for character '9'). Also, because the checks operate on labeled data, the system creates snapshots for each check and attempts to explore additional paths later. For these alternative paths, the string elements are rewritten to be characters that do not represent numbers. In these cases, the while-loop would terminate immediately.

In the example, the program reaches the check on Line 11 after the second loop iteration. Given the original input for *str*, *sum* is 0 at this point and the else-branch is taken. However, because this conditional branch involves the value *sum* that is labeled with l_6, a snapshot of the current program state is created. When this snapshot is later restored, our system needs to rewrite *sum* with the value 82 be able to take the if-branch. To determine how the program state can be updated

consistently, the constraint system is solved for $l_6 = 82$. A solution to this system can be found ($l_0 = 38$, $l_1 = 32$, $l_2 = l_3 = 8$, $l_4 = 80$, and $l_5 = 2$). Using the mappings, this solution determines how the related memory locations can be consistently modified. As expected, *str[0]* and *str[1]* are set to the characters '8' and '2', respectively. The variable *c* is also set to 2.

Non-linear dependencies. The `atoi` function discussed previously represents a more complex example of what can be captured with linear relationships. However, it is also possible that a program performs operations that cannot be represented as linear constraints. These operations involve, for example, bitwise operators such as `and`, `or` or a lookup in which the input value is used as an index into a table. In case of a non-linear relationship, our current system cannot infer the assignment of appropriate values to labels such that a certain memory location can be rewritten as desired. Thus, whenever an operation creates a non-linear dependency between labels l_i and l_j, we no longer can consistently update the state when any label related to l_i or l_j should be rewritten. To address this problem, we maintain a set N that keeps track of all labels that are part of non-linear dependencies. Whenever a label should be rewritten, all related labels are determined. In case any of these labels is in N, the state cannot be consistently changed and the alternative path cannot be explored.

4.2.2 Saving and Restoring Program State

The previous section explained our techniques to track the propagation of input values during program execution. Every memory location that depends on some interesting input has an attached label, and the constraint system determines how values with different labels are related to each other. Based on this information, multiple paths in the execution space can be explored. To this end, our system monitors the program execution for conditional operations that use one (or two) labeled arguments. When such a branch instruction is identified, a snapshot of the current process state is created.

The snapshot of the current execution state contains the content of the complete virtual address space that is in use. In addition, we have to store the current

mappings and the constraint system. But before the process is allowed to continue, one additional step is needed. In this step, we have to ensure that the conditional operation itself is taken into account. The reason is that no matter which branch is actually taken, this conditional operation enforces a constraint on the possible value range of the labeled argument. We call this constraint a *path constraint*. The path constraint has to be remembered and taken into account in case the labeled value is later rewritten (further down the execution path). Otherwise, we might create inconsistent states or reach impossible paths. When the if-branch of the conditional is taken (that is, it evaluates to true for the current labeled value), the condition is directly used as path constraint. Otherwise, when the else-branch is followed, the condition has to be reversed before it is added to the constraint system. To this end, we simply take the condition's negation.

For example, recall the first program that we showed in Figure 4.1. This program uses two checks to ensure that $x > 0$ and $x < 2$ before the `print` function is invoked. When the first if-statement is reached on Line 2, a snapshot of the state is created. Because x had an initial value of 2, the process continues along the if-branch. However, we have to record the fact that the if-branch can only be taken when the labeled value is larger than 0. Assume that the label of x is l_0. Hence, the appropriate path constraint $l_0 > 0$ is added to the constraint system. At the next check on Line 3, another snapshot is created. This time, the else-branch is taken, and we add the path constraint $l_0 >= 2$ to the constraint system (which, because of the else-branch, is the negation of the conditional check $x < 2$). When the process is about to terminate on Line 5, it is reset to the previously stored state. This time, the if-branch on Line 3 must be taken. To this end, we add the path constraint $l_0 < 2$ to the constraint system. At this point, the constraint system contains *two* entries. One is the constraint just added (i.e., $l_0 < 2$). The other one stems from the first check and requires that $l_0 > 0$. When these constraints are analyzed, our solver determines that $l_0 = 1$. As a result, x is rewritten to 1 and the program continues with the call to `print`.

When a program state is restored, the first task of our system is to load the previously saved content of the program's address space and overwrite the current values with the stored ones. Then, the saved constraint system is loaded. Similar to the case in which the first branch was taken, it is also necessary to add the

appropriate path constraint when following the alternative branch. To this end, the path constraint that was originally used is reversed (that is, we take its negation). This new path constraint is added to the constraint system and the constraint solver is launched. When a solution is found, we use the new values for all related labels to rewrite the corresponding memory locations in a consistent fashion. As mentioned previously, when no solution is found, the alternative branch cannot be explored.

Note that at any point during the program's execution, the solution space of the constraint system specifies all possible values that the labeled input can have in order to reach this point in the program execution. This information is important to determine the conditions under which certain behavior is exhibited. For example, consider that our analysis observes an operating system call that should be included into the report of suspicious behavior. In this case, we can use the solution(s) to the constraint system to determine all values that the labeled input can take to reach this call. This is helpful to understand the conditions under which certain malicious behavior is triggered. For example, consider a worm that deactivates itself after a certain date. Using our analysis, we can find the program path that exhibits the malicious behavior. We can then check the constraint system to determine under which circumstances this path is taken. This yields the information that the current time has to be before a certain date.

4.3 System Implementation

We implemented the concepts introduced in the previous section to explore the execution space of Windows binaries. More precisely, we extended our previous malware analysis tool [9] with the capability to automatically label input sources of interest and track their propagation using standard taint analysis (as, for example, realized in [28, 73]). In addition, we implemented the mechanisms to consistently save and restore program states. This allows us to automatically generate more complete reports of malicious behavior than our original tool. The reports also contain the information under which circumstances a particular behavior is observed. In this section, we describe and share implementation details that we consider interesting for the reader.

4.3.1 Creating and Restoring Program Snapshots

Our system (and the original analysis tool) is built on top of the system emulator Qemu [10]. Thus, the easiest way to save the execution state of a program would be to save the state of the complete virtual machine (Qemu already supports this functionality). Unfortunately, when a sample is analyzed, many snapshots have to be created. Saving the image of the complete virtual machine costs too much time and resources. Thus, we require a mechanism to take snapshots of the process' image only. To this end, we developed a Qemu component that can identify the active memory pages of a process that is executing in the guest operating system (in our case, Microsoft Windows). This is done by analyzing the page table directory that belongs to the Windows process. Because Qemu is a PC emulator, we have full access to the emulated machine's physical memory. Hence, we can access the Windows kernel data structures and perform the same calculations as the Windows memory management code to determine the physical page that belongs to a certain virtual address of the process under analysis. Once we have identified all pages that are memory mapped for our process, we simply copy the content of those that are flagged valid. In addition, when creating a snapshot of a process, we have to make a copy of the virtual CPU registers, the shadow memory, and the constraint system.

The method described above has one limitation. We cannot store (or reset) memory that is paged out on disk. This limitation stems from the fact that although we can access the complete main memory from outside, we cannot read the content on the virtual hard disk (without understanding how the Windows file system and swapping is implemented). Thus, we have to disable swapping and prevent the guest OS from moving memory pages to the disk where they can no longer be accessed. In our experiments, we found that this limitation is not a problem as our malware samples had very modest memory demands and never exceeded the 256 MB main memory of the guest OS.

To reset a process such that it continues from a previously saved snapshot, we use a procedure that is similar to the one for storing the execution state. First, we identify all mapped pages that belong to our process of interest. Then, for each page that was previously saved, we overwrite the current content with the

one that was stored. When the pages are restored, we also reset the virtual CPU to its saved state. Note that it is possible that the process has allocated more pages than were present at the time when the snapshot was taken. This is the case when the program has requested additional memory from the operating system. Of course, these new pages cannot be restored. Fortunately, this is no problem and does not alter the behavior of the process. The reason is that all references in the original pages that now point to the new memory areas are reverted back to the values that they had at the time of the snapshot (when the new pages did not exist yet). The only problem is that the newly allocated pages are lost for the process, but still considered in use by the operating system. This "memory leak" might become an issue when, for example, a memory allocating routine is executed various times when different execution paths are explored. Although we never experienced problems in our experiments, one possible solution would be to inject code into the guest OS that releases the memory.

An important observation is that a process can only be reset to a previously stored state when it is executing in user mode. When a process is executing kernel code, reverting it back to a user mode state can leave data structures used by the Windows kernel in an inconsistent state. The same is true when the operating system is executing an interrupt handling routine. Typically, resetting the process when not in user mode leads to a crash or freezes the system.

Our current implementation allows us to reliably reset processes to previous execution states. However, one has to consider the kernel state when snapshots are taken or restored. In particular, we have to address the problem that a resource might be returned to the operating system after a snapshot has been taken. When we later revert to the previously stored snapshot, the resource is already gone, and any handles to it are stale. For example, such a situation can occur when a file is closed after a snapshot is made. To address this problem, we never allow a process to close or free any resource that it obtains from the operating system. To this end, whenever an application calls the `NtClose` function or attempts to return allocated memory to the OS, we intercept the function and immediately return to the user program. From the point of view of the operating system, no handle is ever closed. Thus, when the process is reset to an old state, the old handles are still valid.

4.3.2 Identification of Program Termination

The goal of our approach is to obtain a comprehensive log of the activities of a program on as many different execution paths as possible. Thus, before reverting to a previously stored state, the process is typically allowed to run until it exits normally or crashes. Of course, our system cannot allow the process to actually terminate. Otherwise, the guest operating system removes the process-related entries from its internal data structures (e.g., scheduler queue) and frees its memory. In this case, we would lose the possibility to revert the image to a snapshot we have taken earlier.

To prevent the program from exiting normally, we intercept all calls to the `NtTerminateProcess` system service function (provided by the `ntdll.dll` library). This is done by checking whether the program counter of the emulated CPU is equal to the start address of the `NtTerminateProcess` function. Whenever the inspected process calls this function, we assume that it wishes to terminate. In this case, we can revert the program to a previous snapshot (in case unexplored paths are left).

Segmentation faults (i.e., illegal memory accesses) are another venue for program termination that we intercept. To this end, we hook the page fault handler and examine the state of the emulated CPU whenever a page fault occurs. If an invalid memory access is detected, the process is reverted to a stored snapshot. Interestingly, invalid memory accesses occur relatively frequently. The reason is that during path exploration, we often encounter checks that ensure that a pointer is not null. In order to explore the alternative path, the pointer is set to an arbitrary non-null value. When this value is later dereferenced, it very likely refers to an unmapped memory area, which results in an illegal access.

Often, we encounter the situation that malicious code does not terminate at all. For example, spreading routines are typically implemented as endless loops that do not stop scanning for vulnerable targets. In such cases, we cannot simply end the analysis, because we would fail to analyze other, potentially interesting paths. To overcome this problem, we set a timeout for each path that our system explores (currently, 20 seconds). Whenever a path is still executed when the timeout expires, our system waits until the process is in a safe state and then reverts it to a

previous snapshot (until there are no more unexplored paths left). As a result, it is also not possible for an attacker to thwart our analysis by deliberating inserting code on unused execution paths that end in an endless loop.

4.3.3 Optimization

One construct that frequently occurs in programs are string comparisons. Usually, two strings are compared by performing a sequence of pairwise equality checks between corresponding characters in the two strings. This can lead to problems when one of the strings (or both) are labeled. Note that each character comparison operates on labeled arguments and thus, is a branching point. As a result, when a labeled string of n characters is compared with another string, we create n states. Each of the states $s_i : 0 \leq i \leq n$ represents the case in which the first i characters of both strings match, while the two characters with the offset $i + 1$ differ. For practical purposes, we typically do not need this detailed resolution for string comparisons. The reason is that most of the time, a program only distinguishes between the two cases in which both strings are either equal or not equal. To address this problem, we implemented a heuristics that attempts to recognize string comparisons. This is implemented by checking for situations in which the same compare instruction is executed repeatedly, and the arguments of this compare have addresses that increase by one on every iteration. When such a string comparison is encountered, we do not branch on every check. Instead, we explore one path where the first characters are immediately different, and a second one in which the two strings match. This optimization avoids the significant increase of the overall number of states that would have to be processed otherwise (often without yielding any additional information).

4.3.4 Limitations

In Section 4.3.1, we discussed our approach of never returning any allocated resource to the operating system. The goal was to avoid invalid handles that would result when a process first closes a handle and is then reset to a previous snapshot (in which this handle is still valid). Our approach works well in most cases. How-

ever, one has to consider situations in which a process creates external effects, e.g., when writing to a file or sending data over a network.

There are few problems when a program writes to a file. The reason is that the file pointer is stored in user memory, and thus, it is automatically reset to the previous value when the process is restored. Also, as mentioned previously, files are never closed. Unfortunately, the situation is not as easy while handling network traffic. Consider an application that opens a connection to a remote server and then exchanges some data (e.g., such as a bot connecting to an IRC server). When reverting to a previous state, the synchronization between the application and the server is lost. In particular, when the program first sends out some data, is later reset, and then sends out this data again, the remote server receives the data twice. Typically, this breaks protocol logic and leads to the termination of the connection. In our current implementation, we solve this problem as follows: All network system calls in which the program attempts to establish a connection or sends out data are intercepted and not relayed to the operating system. That is, for these calls, our system simply returns a success code without actually opening a connection or sending packets. Whenever the program attempts to read from the network, we simply return a string of random characters of the maximum length requested. The idea is that because the results of network reads are labeled, our multiple path exploration technique will later determine those strings that trigger certain actions (e.g., such as command strings sent to a bot).

Another limitation is the lack of support for signals and multi-threaded applications. Currently, we do not record signals that are delivered to a process. Thus, when a signal is raised, this only happens once. When the process is later reverted to a previous state, the signal is not resent. The lack of support for multi-threaded applications is not a problem *per se*. Creating a snapshot for the complete process works independently of the number of threads. However, to ensure deterministic behavior our system would have to ensure that threads are scheduled deterministically.

It might also be possible for specially-crafted malware programs to conceal some malicious behavior by preventing our system from exploring a certain path. To this end, the program has to ensure that a branch operation depends on a value that is related to other values via non-linear dependencies. For example, mali-

cious code could deliberately apply non-linear operations such as `xor` to a certain value. When this value is later used in a conditional operation, our system would determine that it cannot be rewritten, as the related memory locations cannot be updated consistently. Thus, the alternative branch would not be explored. There are two ways to address this threat. First, we could replace the linear constraint solver by a system that can handle more complex relationships. For instance, by using a SAT solver, we could also track dependencies that involve bitwise operations. Unfortunately, when analyzing a binary that is specifically designed to withstand our analysis, our prototype will never be able to correctly invert all operations encountered. An example for that are one-way hash functions, for which our system cannot infer the original data from the hash value alone. Therefore, a second approach could be to relax the consistent update requirement. That is, we allow our system to explore paths by rewriting a memory location without being able to correctly modify all related input values. This approach leads to a higher coverage of the code analyzed, but we lose the knowledge of the input that is required to drive the execution down a certain path. In addition, the program could perform impossible operations (or simply crash) because of its inconsistent state. However, frequent occurrences of conditional jumps that cannot be resolved by our system could be interpreted as malicious. In this case, we could raise an appropriate warning and have a human analyst perform a deeper investigation.

Finally, specially-crafted malware programs could perform a denial-of-service attack against our analysis tool by performing many conditional branches on tainted data. This would force our system to create many states, which in turn leads to an exponential number of paths that have to be explored. One solution to this problem could be to define a distance metrics that can compare saved snapshots and merge sufficiently similar paths. Furthermore, we could also treat a sudden, abnormal explosion of states as a sign of malicious behavior.

4.4 Evaluation

In this section, we discuss the results that we obtained by running our malware analysis tool on a set of 308 real-world malicious code samples. These samples were collected in the wild by an anti-virus company and cover a wide range of

malicious code classes such as viruses, worms, Trojan horses and bots. Note that we performed our experiments on all the samples we received, without any preselection.

The 308 samples in our test set belong to 92 distinct malware families (in certain cases, several different versions of a single family were included in the sample set). We classified these malware families using the free virus encyclopedia available at `viruslist.com`. Analyzing the results, we found that 42 malware families belong to the class of email-based worms (e.g., Netsky, Blaster). 30 families are classified as exploit-based worms (e.g., Blaster, Sasser). 10 malware families belong to the classic type of file infector viruses (e.g., Elkern, Kriz). The remaining 10 families are classified as Trojan horses and backdoors, typically combined with bot functionality (e.g., AceBot, AgoBot, or rBot). To understand how widespread our malware instances are, we checked Kaspersky's top-20 virus list for July 2006, the month that we received our test data. We found that our samples cover 18 entries on this list. Thus, we believe that we have assembled a comprehensive set of malicious code samples that cover a variety of malware classes that appear in the wild.

In a first step, our aim was to understand to which extent malware uses interesting input to perform control flow decisions. To this end, we had to define appropriate input sources. In our current prototype implementation, we consider the functions listed in Table 4.1 to provide interesting input. These functions were chosen primarily based on our previous experience with malware analysis (and also based on discussions with experienced malware analysts working in an antivirus company). In the past, we have seen malicious code that uses the output provided by one of these functions to trigger actions. Also, note that adding additional input sources, if required, is trivial and is not a limitation of our approach. During the analysis, we label the return values of functions that check for the existence of an operating system resource. For functions that read from a resource (i.e., file, network, or timer), we label the complete buffer that is returned (by using one label for each byte).

After running our analysis on the complete set of 308 real-world malware samples, we observed that 229 of these samples used at least one of the tainted input sources we defined. The breakdown of the usage based on input is shown in

Interesting input sources	
Check for Internet connectivity	20
Check for mutex object	116
Check for existence of files	79
Check for existence of registry entry	74
Read current time	134
Read from file	106
Read from network	134

Table 4.1: Number of samples that access tainted input sources.

Table 4.1. Of course, reading from a tainted source does not automatically imply that we can explore additional execution paths. For example, many samples copy their own executable file into a particular directory (e.g., the Windows system folder). In this case, our analysis observes that a file is read, and appropriately taints the input. However, the tainted bytes are simply written to another file, but not used for any conditional control flow decisions. Thus, there are no alternative program paths to explore.

Out of the 229 samples that access tainted sources, 172 use some of the tainted bytes for control flow decisions. In this case, our analysis is able to explore additional paths and extract behavior that would have remained undetected with a dynamic analysis only based on a single execution trace. In general, exploring multiple paths results in a more complete picture of the behavior of that code. However, it is unreasonable to expect that our analysis can always extract important additional knowledge about program behavior. For example, several malware instances implement a check that uses a mutex object to ensure that only a single program instance is running at the same time. That is, when the mutex is not found on the first execution path, the malware performs its normal malicious actions. When our system analyzes the alternative path (i.e., we pretend that the mutex exists), the program immediately exits. In such situations, we are only able to increase our knowledge by the fact that the presence of a specific mutex leads to immediate termination. Of course, there are many other cases in which the additional behavior is significant, and reveals hidden functionality not present in a single trace.

Table 4.2 shows the increase in coverage of the malicious code when we explore alternative branches. More precisely, this table shows the relative increase in the number of basic blocks that are analyzed by our system when considering alternative paths. The baseline for each sample is the number of basic blocks covered when simply running the sample in our analysis environment. For a small number of the samples (21 of 172), the newly detected code regions amount to less than 10% of the baseline. While it is possible that these 10% contain information that is relevant for an analyst, they are mostly due to the exploration of error paths that quickly lead to program termination. For the remaining samples (151 of 172), the increase in code coverage is above 10%, and often significantly larger. For example, the largest increase in code coverage that we observed was 3413.58%, when analyzing the `Win32.Plexus.B worm`. This was because this sample only executes its payload if its file name contains the string `upu.exe`. As this was not the case for the sample uploaded into our analysis system, the malware payload was only run in an alternative path. Anecdotal evidence of the usefulness of our system is provided in the following paragraphs, where we describe interesting behavior that was revealed by alternative paths. However, examining the quantitative results alone, it is evident that almost one half of the malware samples in the wild contain significant, hidden functionality that is missed by a simple analysis.

Relative increase	Number of samples
0 % - 10 %	21
10 % - 50 %	71
50 % - 200 %	37
$>$ 200 %	43

Table 4.2: Relative increase of code coverage.

Behavioral analysis results. One interesting class of malicious behavior that can be detected effectively by our system is code that is only executed on a certain date (or in a time interval). As an example for this class, consider the `Blaster` code shown in Figure 4.4. This code launches a denial-of-service attack, but only

after the 15^{th} of August. Suppose that `Blaster` is executed on the 1^{st} of January. In that case, a single execution trace would yield no indication of an attack. Using our system, however, a snapshot for the first check of the if-condition is created. After resetting the process, the day is rewritten to be larger than 15. Later, the system also passes the month check, updating the month variable to a value of 8 or larger. Hence, the multiple execution path exploration allows us to identify the fact that `Blaster` launches a denial-of-service attack, as well as the dates that it is launched.

```
1:  GetDateFormat( LOCALE_409, 0, NULL,
                        "d", day, sizeof(day));
2:  GetDateFormat( LOCALE_409, 0, NULL,
                        "M", month, sizeof(month));
3:
4:  if (atoi(day) > 15 && atoi(month) >= 8)
5:      run_ddos_attack();
```

Blaster Denial-of-Service Attack

Figure 4.4: `Blaster` source code snippet.

Another interesting case in which our analysis can provide a more complete behavioral picture is when malware checks for the existence of a file to determine whether it was already installed. For example, the `Kriz` virus first checks for the existence of the file `KRIZED.TT6` in the system folder. When this file is not present, the virus simply copies itself into the system folder and terminates. Only when the file is already present, malicious behavior can be observed. In such cases, an analysis system that performs a single execution run would only be able to monitor the installation.

Finally, our system is well-suited to identify actions that are triggered by commands that are received over the network or read from a file. An important class of malware that can be controlled by remote commands are IRC (Internet Relay Chat) bots. When started, these programs usually connect to an IRC server, join a channel, and listen to the chat traffic for keywords that trigger certain actions. Modern IRC bots can typically understand more than 100 commands, making a

```
 0:  // receive line from network --> store in array a[]
 1:  // a[0] = command, a[1] = arg1, a[2] = arg2, ...
 2:
 3:  if (strcmp("crash", a[0]) == 0) {
 4:      strcmp(a[5],"crash"); // yes, this will crash.
 5:      return 1;
 6:  }
 7:  else if (strcmp("getcdkeys", a[0]) == 0) {
 8:      getcdkeys(sock,a[2],notice);
 9:      return 1;
10:  }
11:  else if (strcmp("driveinfo", a[0]) == 0) {
12:      DriveInfo(sock, a[2], notice, a[1]);
13:      return 1;
14:  }
```

rxBot Command Loop

Figure 4.5: `rxBot` source code snippet.

manual analysis slow and tedious. Using our system, we can automate the process and determine, for each command, which behavior is triggered. In contrast, when running a bot in existing analysis tools, it is likely that no malicious actions will be seen, simply because the bot never receives any commands. The code in Figure 4.5 shows a fragment of the command loop of the bot `rxBot`. This code implements a series of if-statements that check a line received from the IRC server for the presence of certain keywords. When this code is analyzed, the result of the read from the network (that is, the content of array `a`) is labeled. Therefore, all calls to the `strcmp` function are treated as branching points, and we can extract the actions for one command on each different path.

Performance. The goal of our system is to provide a malware analyst with a detailed report on the behavior of an unknown sample. Thus, performance is not a primary requirement. Nevertheless, for some programs, a significant number of paths needs to be explored. Thus, the time and space requirements for saving and restoring states cannot be completely neglected.

Whenever our system creates a snapshot, it saves the complete active memory content of the process. In addition, the state contains information from the shadow memory and the constraint system. During our experiments, we determined that

the size of a state was equal to about three times the amount of memory that a process has allocated. On average, the size of a state was about 3.5 MB, and it never exceeded 10 MB. The time needed to create or restore a snapshot was 4 milliseconds on average, with a small variance (on an Intel Pentium IV with 3.4 GHz and 2 GB RAM). As mentioned in Section 4.3.2, a timeout of 20 seconds was set for the exploration of each individual program path. In addition, we set a timeout of 100 seconds for the complete analysis run of each sample. This tight, additional time limit was introduced to be able to handle a large number of samples in case certain malware instances would create many paths. In our experiments, we observed that 58% of the malware programs finished before the timeout expired. The remaining 42% of the samples had unexplored paths left when the analysis process was terminated. As a result, by increasing the total timeout, we would expect to achieve an even larger increase in code coverage than that reported in the previous paragraphs. The trade-off is that it would take longer until results are available.

The size of a state could be significantly reduced if we exploited the fact that the majority of memory locations and entries in the shadow memory are 0. Also, we could attempt to create incremental snapshots that only store the difference between the current and previous states. In theory, the number of concurrently active states can be as high as the number of branching points encountered. However, we observed that this is typically not the case, and the number of concurrently active states during the experiments was lower. More precisely, our system used on average 31 concurrent states (the maximum was 469). Note that these numbers also represent the average and maximum depths of the search trees that we observed, as we use a depth-first search strategy. The *total* number of states were on average 32, with a maximum of 1,210. Given the number of concurrently active states, we deemed it not necessary to develop more sophisticated algorithms to create program snapshots. Moreover, in a synthetic benchmark, we verified that our system can handle more than thousand active states.

4.5 Summary

In this chapter, we presented a system to explore multiple execution paths of Windows executables. The goal is to obtain a more comprehensive overview of the actions that an unknown sample can perform. In addition, the tool automatically provides the information under which circumstances a malicious action is triggered.

Our system works by tracking how a program processes interesting input (e.g., the local time, file checks, reads from the network). In particular, we dynamically check for conditional branch instructions whose outcome depend on certain input values. When such an instruction is encountered, a snapshot of the current execution state is created. When the program later finishes along the first branch, we reset it to the previously saved state and modify the argument of the condition such that the other branch is taken. When performing this rewrite operation, it is important to consistently update all memory locations that are related to the argument value. This is necessary to prevent the program from executing invalid or impossible paths.

Our experiments demonstrate that, for a significant fraction of malware samples in our evaluation set, the system is indeed exploring multiple paths. In these cases, our knowledge about a program's behavior is extended compared to a system that observes a single run. We also show for a number of real-world malware samples that the actions that were discovered by our technique reveal important and relevant information about the behavior of the malicious code.

Using the methods presented in both this and the previous chapter, it is possible to generate accurate reports of malicious behavior of malware samples. Even in cases where those samples apply code obfuscation tricks like the ones presented in Chapter 3 or only show their real malicious behavior when very specific conditions are met, the methods presented here will still be able to generate a meaningful profile. Sometimes, however, using only those methods to identify malicious code is not enough to respond to new threads in a timely manner. For example, if a user encounters some malicious binary that none of the analysis systems has had a chance to analyzed yet, he will always be defenseless.

Thus, we want to show in the next chapter how it is possible to identify rogue networks, which are predominantly responsible for the distribution of malicious software. By recognizing those networks, it is possible to generate blacklists and block the download of suspicious binaries from those hosts without going through the time-consuming analysis process first.

Chapter 5

Finding Rogue Networks

Anecdotal evidence indicates the existence of Internet companies and service providers that are under the influence of criminal organizations or knowingly tolerate their activities. Such companies typically control a number of networks with public IP addresses that are abused for a wide range of malicious activities. One such activity is offering bullet-proof hosting, a service that guarantees the availability of hosted resources even when they are found to be malicious or illegal. These hosting services are often used for phishing purposes or for serving exploits and malware. Other malicious activities involve the sending of spam, hosting scam pages, or providing a repository for pirated software and child pornography.

An example of a rogue network that offered bullet-proof hosting was the Russian Business Network (RBN), who made headlines in late 2007 [11, 58]. Various sources alleged that the RBN hosted web sites, exploits, and malware that were responsible for a significant fraction of online scams and phishing. Once publicly exposed, the RBN ceased its operations in St. Petersburg, only to relocate and resume activities in different networks [36]. More recently, a report exposed Atrivo (Intercage), a US-based company that is frequently considered to provide hosting for malicious content [5, 59]. Often referred to as the RBN of the United States, this company is considered to be a "dedicated crime hosting firm whose customer base is composed almost, or perhaps entirely, of criminal gangs" [46]. Shortly after Atrivo made headlines, two more rogue networks, known as McColo and the Triple Fiber Network (3FN), were discovered to be major hosting providers for

malicious content with ties to cybercrime [3, 4, 60]. Again, public outcry quickly lead reputable ISPs to severe their peering relationships with these organizations, effectively cutting them off the Internet.

Obviously, rogue networks and bullet-proof hosting providers are only one component of the flourishing underground economy, which is responsible for many of the security problems that Internet users face. Over the last few years, criminals have increasingly leveraged botnets to hide their tracks [30]. Also, large-scale exploitation (such as the recent wave of SQL injection attacks [38] that affected more than half a million web pages) has lead to a situation where malicious content is unwittingly served by many benign, compromised Internet hosts. These hosts are often combined into fast-flux networks to increase the availability of malicious sites and executables [48].

Despite the large numbers of bot-infected machines and compromised servers, rogue networks do play an important role in the underground economy. These networks often house back-end machines (called motherships) that serve scam pages and exploits, while bots and compromised web pages act as proxies or redirectors. In this setup, a criminal can hide his malicious servers behind a layer of bots, which can be easily replaced when they are taken down or cleaned up [50]. In addition, the content is located at a central location, which eases management. For example, it is straightforward to check for multiple accesses from the same IP. Often, subsequent accesses to malware pages are redirected to benign sites (such as `msn.com`). This makes life more difficult for human malware analysts, but also foils client-side honeypots that require multiple accesses to the same site to determine malicious pages.

In this chapter, we describe *FIRE* (**FI**nding **R**ogue n**E**tworks), a system that monitors the Internet for malicious networks. We believe that it is important to expose such networks, for a number of reasons. First, as the examples of the Russian Business Network, Atrivo, McColo, and 3FN demonstrate, criminals fear public attention. As a result of the increased media coverage, all four networks had to cease their immediate activity. In many cases, it is likely that their operations resumed elsewhere. However, it took some time before the miscreants could restructure their setup, undoubtedly preventing further fraudulent scams and infections during the downtime. Thus, by quickly bringing to light networks that

act maliciously, it becomes more difficult for cyber-criminals to establish a home base. The second advantage of identifying rogue networks is the possibility to generate blacklists that can block all traffic from a netblock, even when certain IPs within this netblock have not yet acted maliciously. This approach prevents criminals from cycling through the available IP space, quickly shifting to a new IP when a current host is blacklisted. Currently, there are manual efforts underway to establish blacklists based on the observation that certain networks are malicious. For example, Spamhaus [95] maintains the *Don't Route Or Peer (DROP)* list, a collection of networks that they consider to be controlled entirely by professional spammers. Spamhaus suggests that traffic from these sources should simply be dropped, and recommends the use of this list by tier-1 ISPs and backbone networks. Another example is the list maintained by EmergingThreats [37], which identifies netblocks that are thought to belong to the Russian Business Network. While such efforts are beneficial, they are expensive and tedious to maintain. Moreover, these lists are often incomplete and limited in scope (for example, limited to spam operations or the RBN in particular). In contrast, *FIRE* operates in an automated fashion, and we aim to capture a broader range of malicious activity, independent of any *a priori* knowledge of criminal organizations.

To identify rogue networks, we rely on a number of data sources that report the malicious actions of individual hosts. Some of the data feeds are publicly available, such as lists of phishing web pages. The other data originates from our own analysis efforts, such as a list of hosts that provide botnet command and control servers and hosts that are found to exploit browser vulnerabilities. Of course, given the widespread use of botnets and the large number of exploited machines, the fact that a host performs malicious actions is no immediate indication that the corresponding ISP or netblock is malicious. Instead, when a host misbehaves, it is possible that attackers were able to compromise and abuse it for nefarious purposes. Thus, it is necessary to search the data for indicators that allow us to distinguish between hosts under the control of rogue (or grossly negligent) ISPs and infected machines of organizations that make a deliberate effort to keep their network clean.

Based on post-processed information obtained from different data sources, we compute a *malscore* (maliciousness score) for individual ASNs (Autonomous Sys-

tem Number). This score quantifies the amount of recent, malicious activity in a network and serves as an indicator for the likelihood that an ASN is linked to cyber-criminals, or at the least, being very negligent in removing malicious content. Using the malscores, it is easy to identify the worst offenders on the Internet and take appropriate actions (such as increasing the public pressure, breaking peering relationships, or putting their IP address space on a blacklist). Moreover, we can track malicious activity over time.

The main contributions of our work are as follows:

- We analyze a number of data sources to identify IP addresses of hosts that misbehave in different ways.
- We present techniques to filter these lists for hosts that likely belong to rogue ISPs. In particular, we combine the information from different data sources to compute a *malscore* that quantifies the malicious activities of an autonomous system.
- We show that our system is successful in identifying a number of rogue ISPs that are known to cooperate with criminal organizations. Moreover, we provide an updated real-time system via the website `maliciousnetworks.org`, which can help to identify rogue ISPs and to assist legitimate ISPs in cleaning up their networks.

5.1 System Overview

The goal of our system is to identify rogue networks. Thus, we first need to concretize what we consider to be a rogue network. Unfortunately, this question is not straightforward to answer. Some service providers are simply lax when it comes to the content that they offer, others are victims of remote exploits, and a few are well-known to blatantly host malicious content. Thus, the fact that a network is the source of unwanted activity does not necessarily qualify it immediately as being malicious.

We consider a rogue network to be a network that is under the control of cyber-criminals or that knowingly profits from cooperating with criminals. Of course,

it is difficult to assert such criminal ties without thorough investigations by law enforcement agencies. Thus, we have to redefine our notion of rogue networks based on the activities that are typically associated with such networks. To this end, we consider a rogue network to be one in which significant malicious activity occurs. In addition, this activity lasts for an extended period of time, regardless of abuse complaints. Our logic behind this is that rogue networks provide hosting for malicious content that often remains up for many days (sometimes even months or years). In contrast, malicious activity in other networks tends to be more short-lived due to abuse reporting and honest attempts to undo the damage.

Of course, there might be cases in which legitimate service providers fail to handle problems in their networks for reasons other than criminal intent (e.g., careless customers, understaffed abuse department). In such cases, these organizations should not immediately be classified as malicious. However, when malicious activity is persistent and ubiquitous in a provider's network, the negligence and failure to act appropriately presents a significant threat to the security of the Internet. As a result, we feel that it is justified to classify such networks as rogue, even though the companies might not directly be affiliated with criminal activities.

Data sources and processing. Given our notion of rogue networks, the basic idea to identify such networks is to check for the presence of a large number of long-lived, misbehaving hosts. To this end, we analyze a number of data sources for IP addresses that have exhibited malicious behavior for an extended period of time (the exact extent of this time span depends on the type of data source and is discussed later).

For our analysis, we utilize three sources of information about malicious activities. One source provides two feeds of hosts that were found to provide command and control services for bots. A second source uses three data feeds to identify servers that were involved in drive-by-download exploits. The third source reports URLs that were found to host phishing pages. We have selected these sources because they are typically associated with malicious activity that is carried out by dedicated machines of rogue networks. Of course, it is easy to incorporate additional sources into our analysis framework.

By sifting through our data sources, the goal is to expose *rogue IPs*. These rogue IPs belong to hosts that have persistently acted in a malicious fashion, and thus, are more likely to belong to rogue networks. This allows us to discard many IP addresses that belong to hosts that were exploited or contaminated by malware, but quickly quarantined. By discarding the large fraction of IPs that exhibit malicious behavior for only a brief period, we avoid the problem of ranking techniques that try to assess the maliciousness of an ISP simply by counting the number of incidents that occur on its network. These approaches often end up with large ISPs in top positions, simply because those ISPs suffer from more compromises due to their large user base. This is despite honest efforts of these ISPs to detect malicious behavior and to contain damage.

Data analysis and malscore computation. Based on the size and the number of active rogue IPs in a network at a certain point in time, we compute a malscore that quantifies the extent of malicious activity in this network. Note that computing malscores is a continuous process. That is, we do not simply obtain a snapshot of rogue networks at a certain point in time, but instead, recompute malscores periodically (currently, once a day). Thus, as new hosts start to perform malicious activity and old hosts cease to be active, the malscores of different networks change. This allows us to monitor and track activity in different networks over time.

We have computed the malscores of networks on the Internet for more than a year. For this period, we found that the malicious activities (and the resulting malscores) are relatively stable for most networks. This indicates, that a number of rogue networks exist that are constantly involved in significant and long-lasting malicious activity. Of course, we also witnessed the interesting rise and decline of certain networks. For example, we could observe the sudden drop of malicious activity associated with Atrivo/Intercage as the network was cut off the Internet by its upstream ISPs. On the other hand, we observed new, malicious networks such as the Novikov Aleksandr Leonidovich autonomous system. This network has only recently appeared in our data, but is already hosting a large number of drive-by-download servers affiliated with the “Beladen” exploit campaign.

The criminals behind these attacks are believed to have close ties to the former RBN [43].

5.2 Data Collection

In this section, we discuss in more detail the three data sources that we use to identify hosts that likely belong to rogue networks. To this end, we first describe, for each data source, how we obtain the IP addresses of hosts that are *actively* engaged in malicious activity.

5.2.1 Botnet Command and Control Providers

Despite the emergence of peer-to-peer-based bots, many botnets still rely on centralized command and control (C&C). For this C&C infrastructure, botmasters typically set up IRC servers that provide channels for bots to join, or web servers that can be periodically polled for new commands. The functioning of the complete botnet depends on the availability of these servers. Thus, a botmaster is interested in hosting his C&C infrastructure on a network where it is safe from takedown.

To identify and monitor the networks affiliated with botnet C&C servers, we utilize data collected from Anubis [2]. Anubis executes the uploaded Windows-based malware binaries in a virtual environment and records file system and registry modifications, process information, and network communications. We are particularly interested in the network traffic (if any) generated by the malware.

IRC-based botnets. When Anubis monitors IRC traffic the corresponding nickname, server, and channel information is logged. This IRC information is almost always associated with botnet C&C traffic. To monitor whether IRC C&C channels are active, we use a custom IRC client that leverages the recorded credentials to connect to the IRC server and join the channel. Because we are primarily interested in the longevity of the C&C server, we resolve the C&C server's host name to one or more IP addresses, and then connect to each IP at regular intervals. When the C&C server is not identified by a DNS name but by an IP address, then this

address is used directly. A host (an IP address) is considered to be active when our client can join the corresponding C&C channel. Sometimes, transient network problems prevent us from connecting to a host. In such cases, it would be undesirable and premature to declare a host as inactive. Thus, we require that an active C&C channel is unreachable for two days before declaring the corresponding IP address as inactive.

Interestingly, in a number of cases, we observed that a channel (and the corresponding server) was reachable, but no malicious activity was noticeable. This is frequently the case when a bot channel is created on a well-known IRC network (such as `undernet` or `efnet`). The reason is that the IRC administrators of these networks quickly ban the botmaster and remove the channel. However, subsequent logins from bots or other users reopen the channel, thus making the channel available and leaving the impression that it is still active. To mitigate this problem, we modify our approach to determine whether a botnet C&C host is active. More precisely, in addition to the requirement that a server is reachable and the appropriate channel exists, we also require that the channel shows bot-related activity. To this end, we introduce heuristics that check the messages and channel topics for well-known IRC bot commands (such as *download*, *update*, *dos*) and signs of encoded or encrypted commands. A channel is considered up only when such indicators are present.

HTTP-based botnets. To identify and monitor web-based botnet C&C servers from samples collected by Anubis, we first require a mechanism to distinguish between legitimate HTTP traffic and traffic related to botnet commands. This is necessary because HTTP traffic sent by a malware sample does not immediately imply a connection to a C&C server (HTTP connections are often used to check for network connectivity or download updates). To identify HTTP C&C traffic, we manually define static, malicious characteristics (signatures) of requests used by well-known botnets. These characteristics include content from the HTTP request path and parameters, HTTP headers and POST data, and the HTTP response from the web server. Such static features are useful even for botnets that use encryption because they frequently send an encryption key, bot identifier, version number,

and other parameters to the web server. Thus, the HTTP C&C server must know how to parse the request in a specific format.

As an example of a web-based botnet that we have been monitoring, consider Pushdo/Cutwail, which is believed to be one of the largest, active botnets used for spam. When a Cutwail bot connects to the C&C server, it will often request one or more executables. Although the botnet utilizes encryption, the request path for these binaries contains a predictable semi-static format, such as the prefix `/40E8`. The response from the web server contains one or more executables typically around 100KB. Currently, we are monitoring 24 different types of web-based botnets including Coreflood, Torpig, and Koobface. The architecture for monitoring botnet C&C servers is shown in Figure 5.1.

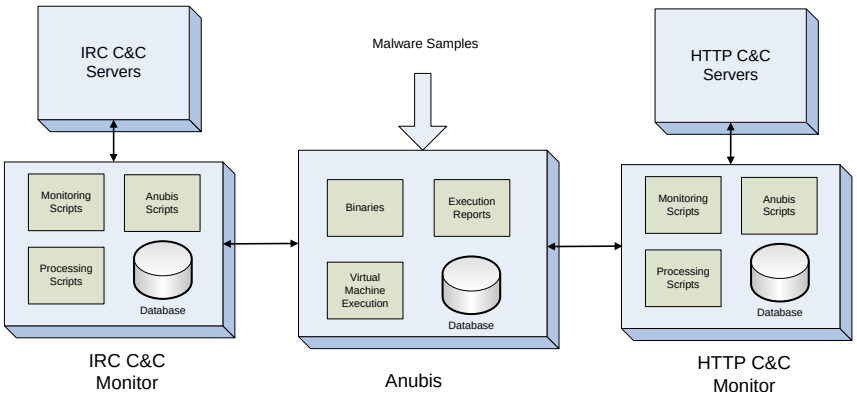


Figure 5.1: Architecture for C&C botnet monitoring

5.2.2 Drive-by-Download Hosting Providers

Our second data source is a list of servers that host malware executables distributed through drive-by-download exploits. Drive-by-downloads are a means of malware distribution where executables are automatically installed on victim machines without user interaction. Typically, the only requirement is for a user to visit a web page that contains an exploit for her vulnerable browser. In some cases, the exploit and the malware executable is hosted on a compromised host, while in other cases, a compromised web page is only used to redirect the victim

to a second machine that performs the exploit (often referred to as a mothership). These mothership servers are frequently located in rogue networks.

There are three data feeds that we use to identify drive-by-download servers. The first feed is through Wepawet [109], a system that checks user-submitted web pages (URLs) for malicious Javascript. In particular, we are interested in cases where malicious script contains shellcode that downloads and executes malware. When malware is discovered, Wepawet records the locations of these binaries and exports them to *FIRE*. The second data feed is through a daily compilation of URLs found in spam mails that are caught in the spam traps of a computer security company and an Internet Service Provider. The third feed is a daily-updated list of "spamvertised" URLs (advertised via spam) provided by Spamcop [94]. So far, after eliminating duplicates, we have recorded more than 1.2 million spamvertised links. Of course, not every URL in a spam email points to a site that launches a drive-by exploit. Instead, these URLs frequently lead to shady businesses such as online pharmacies, casinos, or adult services. To identify those sites and pages that actively perform drive-by-exploits, we use the Capture Honeypot Client (HPC) [90]. Capture is able to find web-based exploits by opening a potentially malicious web site in a browser on a virtual machine. After visiting a page, the state of the virtual machine is inspected and suspicious changes (i.e., the creation of new files or the spawning of new processes) are recorded, as they indicate that the guest system was compromised by a web-based exploit.

For our analysis, we use a total of eight virtual machines (VMs) dedicated to scanning web pages. All VM images are running Windows XP Professional (Service Pack 2), without any patches installed and automatic updates disabled. To catch recent exploits, we have installed the Flash and Quicktime plug-ins.

When the Capture honey client is compromised by visiting a certain URL, we inspect the network traces recorded from Capture HPC. We are not interested in the server that hosts the web site that contains an exploit. We have observed that those machines are often legitimate web servers that are victims of compromise and, therefore, do not yield much information about malicious networks. Thus, if the malicious binary that is part of an exploit is downloaded from the same server, we ignore that host for our analysis. In the more interesting case, an exploit has been injected into a web page and the associated binary is hosted on a

different machine (mothership server that usually serves binaries for many different exploits). Due to the importance of this mothership servers for the criminals behind the exploit, these machines are often located in malicious networks where the chance that it is being shut down is low. Thus, we only consider the IP addresses of those mothership servers for our analysis. Once we have discovered a download server, we revisit it once per day to assess its uptime. When a host cannot be reached for more than two weeks, it is considered to be inactive and, thus, removed from the analysis.

5.2.3 Phish Hosting Providers

The third data source to identify rogue networks is derived from information about servers that host phishing pages. Typically, phishing pages are set up to steal login credentials, credit card numbers, or other personal information. Often, these pages are hosted on compromised servers and are taken down quickly. To mitigate this problem, phishers often resort to hosting their phishing pages directly in networks where there is little or no control of the offered content.

To locate phishing sites, we use an XML feed provided by PhishTank [77]. Once a day, this feed provides our system with URLs of phishing pages that are verified by the PhishTank community. Interestingly, all URLs on the PhishTank list are considered to be online. However, our experiments have shown that phishing pages are often taken offline so quickly that the list is already outdated after one day.

To compute the status of phishing IPs, we attempt to download the web page located at a given phishing URL once per day. This is done until either the domain (of the URL) can no longer be resolved, or the site is offline for more than one week. A phishing site is considered offline by our system when the web server is not reachable anymore or when the phishing page has been replaced by another page that is not a phish (usually a HTTP 404 error page or a phishing warning page).

5.3 Data Analysis

In this section, we discuss our techniques to identify rogue networks and compute their malscores based on the analysis of the individual data sets that we collect.

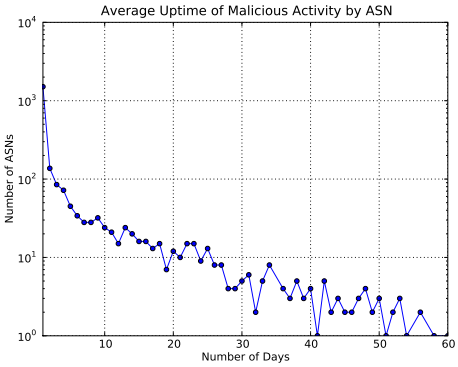
5.3.1 Longevity of Malicious IP Addresses

The primary characteristic that distinguishes between rogue and legitimate networks is the longevity of the malicious services. Most legitimate networks are able to clean up illicit content within a matter of days. In contrast, we have observed malicious content that has been online for the entire monitoring period of more than a year. Figure 5.2a shows the average uptime of malicious IPs per ASN. It can be seen that the vast majority of networks remove the offending content in less than 10 days. However, there were 361 ASNs that had hosts with an average lifespan of more than 10 days in our feeds. Also, we discovered that each type of malicious activity displays different behaviors and average uptime.

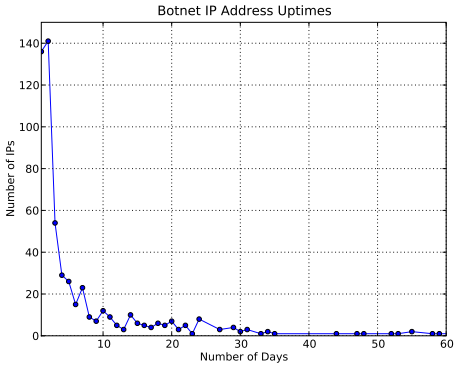
Since May 2008, we have observed botnet C&C servers on 1,269 IP addresses. Figure 5.2b displays the uptime of the botnet C&C servers from 0-60 days. Note that we observed C&C servers that were online for more than 60 days, but limited the x-range of the graph to illustrate the rapid decline in botnet C&C servers that are taken down after only a few days, mainly by reputable IRC and web hosting providers.

We have been monitoring 1,161 of drive-by-download servers since August 2008. These servers have a much higher average lifetime than the other sources depicted in Figure 5.2c. In fact, the number of drive-by-download servers that have been online for more than 60 days is 92, or more than 15%. Also, there have been 17 (approximately 3% of all) drive-by-download servers that have been online since the start of our collection.

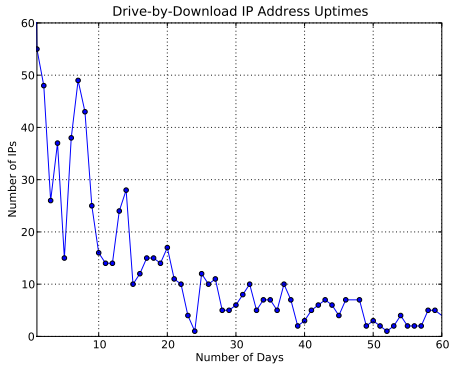
From July 2008, we recorded 12,149 IP addresses hosting phishing websites. Similar to botnet C&C servers, the majority of phishing websites were online for only a few days. However, we also observed a few phishing sites that were online for more than a year. Figure 5.2d shows the uptime for the first 60 days for phishing hosts.



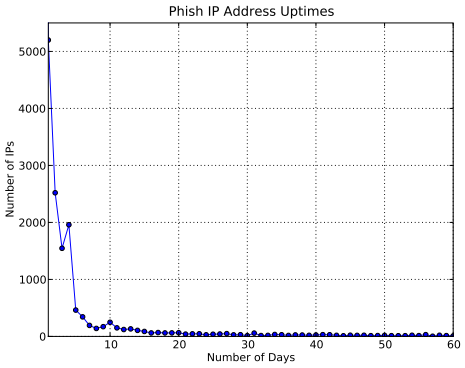
(a) Average IP uptime by ASN.



(b) Botnet uptime between 0-60 days.



(c) Drive-by uptime between 0-60 days.



(d) Phishing uptime between 0-60 days.

Figure 5.2: Uptimes for different sources

As mentioned previously, we use the longevity of malicious services as a distinguishing feature of rogue networks. This insight is supported by the previously-shown data, which demonstrates that a small number of ISPs is responsible for most persistent, malicious activity. To discard IPs that have been active for a short time only, we introduce a threshold δ. IP addresses that are active less than this threshold are not considered rogue and discarded from the subsequent malscore computation. This removes most of the phishing pages that are hosted on free web spaces or hacked machines, and legitimate IRC/web servers that are temporarily abused for botnet communications. As we will explain later in more detail (in Section 5.4.3), we do not use a threshold-based filter for drive-by-download servers. The reason is that such servers are difficult to set up, and, thus, are typically a

direct indication for rogue networks. This is also reflected in the uptime graph for drive-by download servers (Figure 5.2c), which is different than for the other two data sources.

The output of the filtering step (which removes short-lived botnet C&C and phishing IPs) is a list of active, rogue IPs that constitute the input to the malicious score computation process, which is discussed in the next section. In Section 5.4.3, we will come back to the effects of selecting different values for the threshold δ on the overall ASN ranks.

5.3.2 Malscore Computation

Once per day, the data collection process produces three lists $\mathcal{L}_i$ of active, rogue IPs (each derived from a different data source i). In the next step, the goal is to combine this information to expose organizations that act maliciously. For this, we consider an organization to be equivalent with an autonomous system (AS). An autonomous system is a group of a single entity (RFC 1771). Thus, it is a natural choice to perform analysis at the AS-level.

To identify those autonomous systems that are most likely malicious, we first map all IP addresses on the three lists to their corresponding ASN. For this, we query the `whois` database, selecting the most specific entry for an IP address in case multiple autonomous systems announce a particular IP. We are aware that the `whois` data might not be completely accurate. However, even in case of small errors, the database is sufficiently complete and precise to recognize the worst evildoers.

A straightforward approach to identify those autonomous systems that are most malicious is to compute, for each AS, the sum of the IPs on the three lists that belong to this AS. While simple, this technique is not desirable because it ignores the size of a network. Clearly, when an AS P controls many more live hosts than AS Q, we can expect that the absolute number of malicious hosts in P are higher than in Q, even though the relative numbers might show the opposite. To avoid this pitfall, we compute the maliciousness score (malscore) $\mathcal{M}_A$ for an AS P as follows:

$$\mathcal{M}_P = \rho_P * \sum_{i=1}^{3} n_i(P) \tag{5.1}$$

In Equation 5.1, $n_i(P)$ is the number of IP addresses on list $\mathcal{L}_i$ that belong to the autonomous system P. Moreover, the malscore for each AS is adjusted by a factor ρ, which is indirectly proportional to the number of hosts in a network. That is, ρ decreases for larger networks.

The purpose of ρ is to put into relation the number of incidents with the number of active hosts in an autonomous system. This requires, for each AS, the knowledge of the number of live (active) hosts that are operating in the networks of this AS. Clearly, this knowledge is difficult to obtain precisely, and it also can change over the course of several months. Previous work attempted to address this question [81], resorting to the idea of sending ping probes to a well-chosen subset of the IP addresses of a network. While these techniques can discriminate well between completely inactive (dark) regions and used networks, it is still quite difficult to determine the exact number of active hosts. Also, it is possible that networks are configured so that they do not respond to ping requests at all, thereby skewing the results. For these reasons, we decided to estimate the size of a network based on the size of the networks (i.e., the number of IP addresses) that an AS announces as routeable to the global Internet. To determine the size of the address space that an AS announces to the Internet, we leverage data provided by the Cooperative Association for Internet Data Analysis (CAIDA). CAIDA is a collaborative undertaking among organizations in the commercial, government, and research sectors that promotes cooperation in the engineering and maintenance of a robust, scalable, global Internet. In this role, CAIDA makes available a variety of data repositories that provide up-to-date measurements of the Internet infrastructure. One of these data repositories [51] shows a ranking of autonomous systems based on the size of their customer cones (address spaces). This information is compiled from RouteViews BGP tables.

We define $size_p$ as the number of /20 prefixes that an AS P announces. With this, we define ρ as shown in Equation 5.2 below. As desired, ρ decreases when $size_p$ increases.

$$\rho_p = 2^{-size_p/c}, \;\; where \;\; c = 4 \tag{5.2}$$

Of course, we are aware of the fact that the announced address space is not a perfectly reliable indicator for the number of active hosts. For example, there are network telescopes or educational institutions such as MIT that announce huge address ranges while having few or no live hosts. However, such networks are infrequent and, given the shortage of available IPv4 address space, many networks densely populate their available space. On the other hand, masquerading (network address translation - NAT) might result in multiple hosts sharing a singe IP address. Because of the imprecision that is inherent in estimating the number of active hosts, we limit the impact of $size$ on ρ by a parameter c. Empirically, we found that a value of $c = 4$ yields god results. In Section 5.4.3, we motivate this choice and discuss the influence of different values of c on our results.

5.4 Evaluation

In this section, we analyze the quality of our results and discuss interesting findings. Moreover, we discuss in more detail the choice of important system parameters (such as the time threshold δ and size parameter c).

5.4.1 Analysis Results and Malicious Networks

Table 5.1 shows a snapshot of our system on June 1st, 2009, listing the ten entries with the largest malscores and the originating country (using the `ip2location.com` database). For this snapshot, we computed the maliciousness scores for all 417 autonomous systems that control at least one active, rogue IP.

Unfortunately, we do not have ground truth available that would allows us to evaluate the results of our system in a quantitative fashion. In fact, if such information would be available, then there would be no need for our system. Thus, we can only argue qualitatively that our system produces meaningful and interesting insights into the behavior of rogue networks. We can also observe how events on

the Internet (such as shutting down a rogue ISP) are reflected in the malscores of different networks.

#	ASN	Name	Country	Score	S	G	Z	Blogs
1	23522	GigeNET	US	42.4	1	-	-	
2	44050	Petersburg Internet Network	UK	28.0	-	-	6	[34]
3	3595	Global Net Access	US	18.2	-	23	-	
4	41665	National Hosting Provider	ES	16.5	-	104	5	
5	8206	JUNIKNET	LV	14.1	-	30	-	
6	48031	Novikov Aleksandr Leonidovich	UA	14.0	-	-	-	[43]
7	16265	LEASEWEB	NL	13.0	24	14	-	
8	27715	LocaWeb Ltda	BR	11.6	-	130	-	
9	22576	Layered Technologies	US	11.5	-	64	-	[33]
10	16276	OVH OVH	FR	10.6	25	18	-	

S = Shadowserver, G = Google Safebrowsing, Z = ZeusTracker

Table 5.1: *FIRE* Top 10 for June 1st, 2009

Correctness of results. The top ten autonomous systems reported by *FIRE* on June 1st host a large number of persistent, malicious servers. In an attempt to confirm that our results are correct and meaningful, we leveraged a number of third party efforts that attempt to track down certain types of malicious activity on the Internet. More precisely, we first obtained a top-25 list that is complied by the ShadowServer Foundation [92] that shows the most malicious networks with regards to botnet activity. Then, we looked at Google's Safe Browsing initiative [53] and extracted the top 150 ASNs, based on the absolute numbers of malicious drive-by servers that Google identified. In addition, we used the top-10 entries provided by ZeusTracker [115], a network that monitors and lists command and control servers for the Zeus botnet. Finally, we searched a number of blogs written by well-known security researchers for references to malicious and rogue ISPs and networks.

For each of our top ten entries, we then tried to find evidence in any of the third party lists that would confirm that a network is known to be rogue, or at least, strongly linked to certain malicious activities. Table 5.1 shows that we were successful for all ten entries.

ASN	Name	FIRE Rank	Large Network
AS23522	GigeNET	1	
AS3265	XS4ALL	118	X
AS25761	Staminus Comm	-	
AS30058	FDCservers.net	-	
AS174	Cogent	148	X
AS2108	Croatian Research	-	
AS31800	DALnet	-	X
AS13301	Unitedcolo.de	86	
AS790	EUnet Finland	-	
AS35908	SWIFT Ventures	68	

Table 5.2: ShadowServer Botnets Top 10 for June 1st, 2009

ASN	Name	FIRE Rank	Large Network
AS4134	Chinanet Backbone No.31	17	X
AS21844	ThePlanet.com	13	
AS4837	China169 Backbone	90	X
AS36351	SoftLayer Technologies	30	
AS26496	GoDaddy.com	15	X
AS41075	ATW Internet Kft.	23	
AS4812	Chinanet-SH-AP Telecom	89	X
AS10929	Netelligent Hosting	12	
AS28753	Netdirect	11	
AS8560	1&1 Internet AG	-	X

Table 5.3: Google Safe Browsing Top 10 for June 1st, 2009

In our list, IPNAP-ES (GigeNET) has consistently ranked among the top malicious network, because it hosts the largest numbers of IRC botnet C&C servers. This is confirmed by the findings of ShadowServer. Some security forums have actually reported botnet activity from IPNAP as early as 2006. The Petersburg Internet Network (PIN), currently ranked second in Table 5.1, is known to be hosting the Zeus malware kit (also known as Zbot and WSNPoem). On the statistics page of the ZeusTracker [115], PIN's network is ranked first on the list of Zeus hosts as of June 1st, 2009. As the table also shows, as many as seven out of the ten ASNs

ZeusTracker			
ASN	Name	FIRE Rank	Large Network
AS21844	ThePlanet.com	13	
AS12695	Digital Network JSC	16	
AS9800	China Unicom	28	X
AS6849	JSC UkrTelecom	-	
AS41665	National Hosting Provider	4	
AS44050	Petersburg Internet Network	2	
AS43689	Dankon Ltd.	-	
AS35118	SmartLogic Ltd	44	
AS9394	China Railway Internet	88	X
AS4645	HKNet Co. Ltd	-	

Table 5.4: ZeusTracker Top 10 for June 1st, 2009

are ranked highly by Google's malicious site analysis. Although the ZeusTracker only covers one specific botnet, there is overlap with four out of the top 15 entries in *FIRE*, including PIN, National Hosting Provider, ThePlanet (rank 13 in *FIRE*) and Digital Network (rank 15 in *FIRE*).

It is also interesting to note that the "Novikov Aleksandr Leonidovich" AS has been linked to the recent Beladen drive-by-download exploit campaign [43], which is believed to be run by the same criminals that operated the Russian Business Network.

From the perspective of these three independent data sources, we cover nearly all of the most malicious networks. In many cases, larger networks are given an unfair bias in these lists due to the number of compromised hosts on their network. As a result, we tagged these large networks with an X in each table to show that they are false positives.

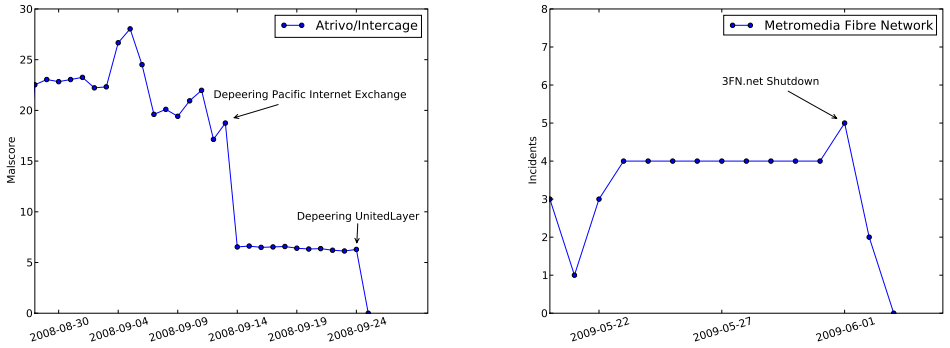
5.4.2 Interesting (Historic) Malscore Changes

A number of networks ceased to engage in malicious activity over the last months (during the continuous monitoring period). The most prominent was Intercage (Atrivo), a network that was considered to be the American RBN equivalent. It hosted nearly every form of malicious and illegal content, and it was ranked fourth on our list in September 2008. In August, Intercage attracted considerable atten-

tion from the Internet community when a white paper was published documenting malicious activities originating from their network [5]. Consequently, two of their transit providers, Pacific Internet Exchange (PIE) and UnitedLayer, depeered with them, cutting them off from the rest of the Internet. Figure 5.3a shows how *FIRE* recorded those incidents. Until mid-September, Atrivo was given a very high score. Then, the de-peering of PIE shut off many malicious hosts, resulting in a considerably lower score. Finally, the second de-peering, about a week later, removed Atrivo completely from the list.

Another success story in the fight against Internet crime happened recently, when the U.S. Federal Trade Commission (FTC) shut down the web hosting company Triple Fiber Network (3FN.net) [60]. The network of 3FN was known to shelter servers that were engaged in all kinds of malicious activities, especially hosting command and control servers for the Cutwail botnet. Therefore, on June 1st, 2009, the FTC decided to have 3FN disconnected from the Internet. As expected, this takedown was immediately noticeable in the *FIRE* data. 3FN.net partnered with several autonomous systems including the Metromedia Fiber Network. The graph in Figure 5.3b clearly demonstrates the effectiveness of Metromedia breaking their ties to 3FN after June 1st.

These two incidents clearly highlight the benefits and importance of exposing disreputable hosting and transit providers, which is the ultimate goal of the *FIRE* system.



(a) Intercage (b) Metromedia Fiber Network

Figure 5.3: De-peering effects.

5.4.3 Sensitivity of Important Parameters

Now that we have presented our results, we will explain the threshold parameters to filter reputable networks.

Longevity thresholds. In order to distinguish between rogue and benign networks, *FIRE* uses a threshold based on the longevity of a server. If a malicious host is online for more than the time given as this threshold, the IP is flagged as malicious. If the server is taken offline before it reaches the threshold, *FIRE* discards the host in the scoring phase. The choice of this threshold is of vital importance for the functioning of the analysis. If the threshold is selected too low, there will be a lot of compromised servers considered as part of malicious networks, which skews the analysis. If it is chosen too high, many malicious servers will be missed.

To show the influence of short-lived servers on the data collected by *FIRE*, we introduce a distance metric on the rankings of malicious networks. In order to compare two lists A and B of offending ASNs sorted by number of incidents, we count how many ASNs have a different position in A and B and add to that value the number of ASNs that have a different incident count in A and B. This metric gives a quantification of how much two lists of worst offenders differ.

In our approach, we used this metric to obtain suitable threshold values. First, we calculated top offending network rankings for a small threshold value. Then, we iteratively increased the threshold by a small value and recalculated the ranking. With the use of the metric above, we were able to determine, how much the small threshold change influenced the resulting lists of malicious networks. We applied this procedure to the three data sources phishing servers, botnet C&C servers and drive-by-download servers. For every day since January 1st, 2009 we calculated the change in rankings and averaged the results. Figure 5.5 shows the resulting differences. The Figures 5.5a and 5.5b indicate that for phishing servers and botnet control servers there is a lot of fluctuation when threshold values are low. This is a direct result of the fact that these data sources contain many servers that are taken offline after only one or two days. Thus, we select the thresholds in a way that these servers are ignored. An ideal threshold value should be chosen

high enough that the spike at the beginning of those graphs are cut off and the fluctuations around the threshold are low. Thus, a threshold value that lies to the right of the initial peak in the curve is the optimal choice. Currently, *FIRE* uses the daily thresholds $\delta_{phish} = 3$ and $\delta_{bot} = 4$.

For drive-by-download servers, we could not observe such behavior. Figure 5.5c shows a constant fluctuation if we remove servers with low uptime. The reason that drive-by-download servers are not taken offline quickly is that setting up drive-by downloads is a relatively complex task. These servers are often set up by professional criminal organizations who do not want to risk that their exploits fail because the mothership server, which hosts the executable, is taken offline. Thus, the only data source that we do not take uptime into account is drive-by-download servers because an overwhelming majority are hosted on malicious networks.

Size parameter. One feature of *FIRE* is the elimination of large networks from the list of true malicious ASNs if those networks have a larger absolute number of malicious hosts, but relatively few rogue hosts. As described in Section 5.3.2, our computation uses a parameter c to scale the malscore of larger networks. We are interested in choosing a value for c that is large enough to reduce the rank of larger networks, while not excluding them completely.

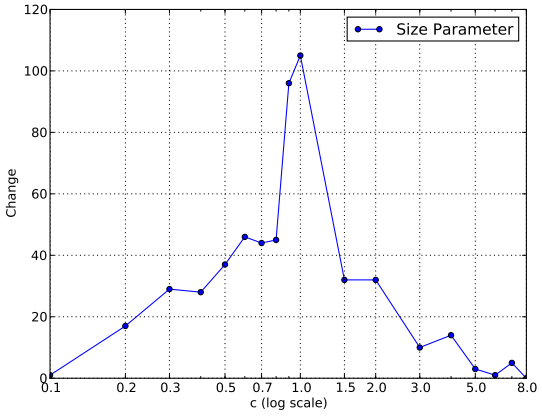


Figure 5.4: Sensitivity of Parameter c.

To show the effects of various choices for the parameter c, we calculated the lists of top offending networks for varying parameter values. Again, we use the metric presented above to show how small changes of c influence the ASN rankings. The changes we obtained are shown in Figure 5.4. For c values less than 1, the overall rank changes are minimal. This is due to the fact that with very small values for c, the resulting lists are only sorted by ASN size, regardless of the number of incidents. Similarly for larger values of c greater than 1, the rankings are primarily sorted by incident count.

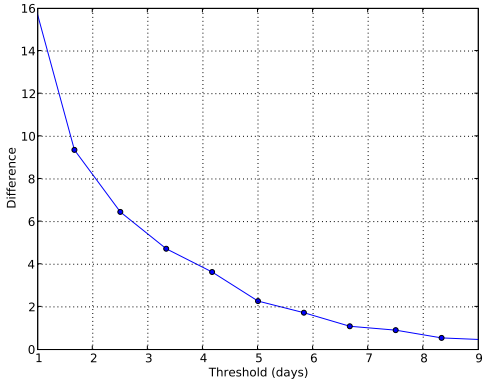
For our analysis, it is thus important to choose a value for c that is located on the right side of the peak shown in the graph, as we want to favor incident count over network size. Also, it is important that the network size does not have a significant influence on the rating. Thus, we want to select an area of the graph where ranking fluctuations low. This lead to the choice of the threshold $c = 4$ for the malscore computation.

5.5 Summary

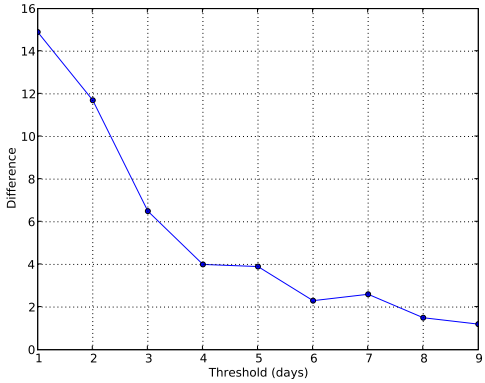
In this chapter, we presented *FIRE*, a novel system to automatically identify and expose organizations and ISPs that demonstrate persistent, malicious behavior. *FIRE* can help isolate networks that tolerate and aid miscreants in conducting malicious activity on the Internet. It does this by actively monitoring different data sources such as botnet communication channels, drive-by-download servers, and feeds from phishing web sites. Because it is important to distinguish between networks that are knowingly malicious and networks that are victims of compromise, we refine the collected data and correlate it to deduce the level of maliciousness for the identified networks. Our ultimate aim is to automatically generate results that can be used to pinpoint and track organizations that support Internet miscreants and to help report and prevent criminal activity. Furthermore, the networks we identify can also be used by ISPs as blacklists in order to simply block traffic that is originating from them. Hence, an ISP can enhance the security of its users by not allowing malicious traffic to reach them.

Although much work has been done on studying malicious activity on the Internet (such as phishing, drive-by-download exploits, and malware-based scams),

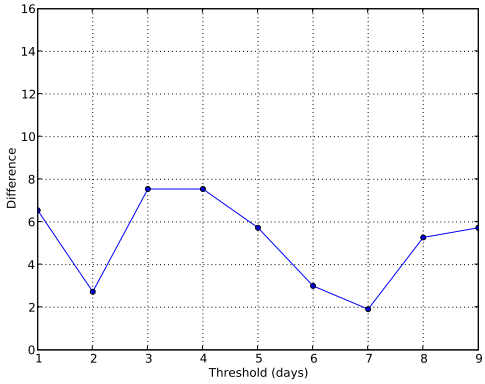
not much focus has been put on automatically identifying the networks and infrastructures used by the attackers. With the novel work we present in this paper, we approach the problem from a different angle and hope to help prevent victims from accessing or receiving traffic from networks that have proven to be malicious in nature.



(a) Phishing Servers



(b) Botnet Servers



(c) Download Servers

Figure 5.5: Ranking changes for varying thresholds.

Chapter 6

Related Work

Malicious Code Analysis. Analyzing malicious executables is not a new problem; consequently, a number of solutions already exist. These solutions can be divided into two groups: *static analysis* and *dynamic analysis* techniques. Static analysis is the process of analyzing a program's code without actually executing it. This approach has the advantage that one can cover the entire code and thus, possibly capture the complete program behavior, independent of any single path executed during run-time. In [17], a technique was introduced that uses model checking to identify parts of a program that implement a previously specified, malicious code template. This technique was later extended in [18], allowing more general code templates and using advanced static analysis techniques. In [61], a system was presented that uses static analysis to identify malicious behavior in kernel modules that indicate a rootkit. Finally, in [57], a behavioral-based approach was presented that relies heavily on static code analysis to detect Internet Explorer plug-ins that exhibit spyware-like behavior. The main weakness of static analysis is that the code analyzed may not necessarily be the code that is actually run. In particular, this is true for self-modifying programs that use polymorphic or metamorphic techniques [101] and packed executables that unpack themselves during run-time [76].

Because of the many ways in which code can be obfuscated and the fundamental limits in what can be decided statically, we firmly believe that dynamic analysis is a necessary complement to static detection techniques. In [12], a

behavior-based approach was presented that aims to dynamically detect evasive malware by injecting user input into the system and monitoring the resulting actions. In addition, a number of approaches exist that directly analyze the code dynamically. Unfortunately, the support for dynamic code analysis is limited; often, it only consists of debuggers or disassemblers that aid a human analyst. Tools such as Anubis[2], CWSandbox [110], the Norman SandBox [75], or Cobra [105] automatically record the actions performed by a code sample, but they only consider a single execution path and thus, might miss relevant behavior. To address this limitation and to capture a more comprehensive view of a program's behavior, we developed our approach to explore multiple execution paths.

A very recent work that addresses the detection of hidden, time-based triggers in malware is described in [29]. In their work, the authors attempt to automatically discover time-dependent behavior by setting the system time to different values. The problem is that time-based triggers can be missed when the system time is not set to the exact time that the malware expects. In our approach, we do not attempt to provide an environment such that trigger conditions are met, but explore multiple code paths independent of the environment. Thus, we have a better chance of finding hidden triggers. In addition, our approach is more comprehensive, as we can detect arbitrary triggers.

Finally, in [13], the authors present a system that is similar to ours in its goal to detect trigger-based malware behavior. The main differences are the system design, which is based on mixed execution of binary code using elements of symbolic execution, and a less comprehensive evaluation (on four malware samples).

Software testing. The goal of our work is to obtain a more complete picture of the behavior of a malicious code sample, together with the conditions under which certain actions are performed. This is analogous to software testing where one attempts to find inputs that trigger bugs.

A number of test input generation systems [14, 44, 45] were presented that analyze a program and attempt to find input that drives execution to a certain program point. The difference to our approach is that the emphasis of these systems is to reach a certain point, and not to explore the complete program behavior. Other tools were proposed that explore multiple paths of a program to detect implemen-

tation errors. For example, model checking tools [26, 47, 49] translate programs into finite state machines and then reason whether certain properties hold on these automata. The systems that are closest to our work are DART [42] and EXE [15]. Both systems use symbolic execution [56]. That is, certain inputs are expressed as symbolic variables, and the system explores in parallel both alternative execution paths when a conditional operation is found that uses this symbolic input. Similar to our approach, these systems can explore multiple execution paths that depend on interesting input. Also, the conditions under which certain paths are selected can be calculated (and are subsequently used to generate test cases). The main differences to our technique are the following. First, the goal of these systems is to explore programs for program bugs while our intent is to create comprehensive behavioral profiles of malicious code. Second, we do not have the possibility of using source code and operate directly on hostile (obfuscated) binaries. This leads to a significantly different implementation in which interesting inputs are dynamically tracked by taint propagation. Also, the problem we are addressing is complicated by the fact that we are not able to utilize built-in operating system mechanisms (e.g., fork) to explore alternative program paths. Hence, we require an infrastructure to save and restore snapshots of the program execution.

Speculative execution. In [74], a system was presented that uses process snapshots to implement speculative execution. In distributed files systems, processes typically have to wait until remote file system operations are completed before they can resume execution. With speculative execution, processes continue without waiting for remote responses, based on locally available data only. When it later turns out that the remote operation returns data that is different from the local one, the process is reset to its previously stored snapshot. When no changes are present, the process can continue, and the system is successful in masking a slow I/O operation. The concept of snapshots used in speculative execution is similar to the one in our work. The difference is that we use snapshots as a means to explore alternative execution paths, which requires consistent memory updates.

Code obfuscation. The two areas that are most closely related to our work on the limits of static malware analysis are code obfuscation and binary rewriting.

Code obfuscation describes techniques to make it difficult for an attacker to extract high-level semantic information from a program [21, 108]. This is typically used to protect intellectual property from being stolen by competitors or to robustly embed watermarks into copyrighted software [20]. Similar to our work, researchers proposed obfuscation transformations that are difficult to analyze statically. One main difference to our work is that these transformations are applied to source code. Source code contains rich program information (such as variables, types, functions, and control flow information) that make it easier to apply obfuscating operations.

In [22], mechanisms are proposed that operate by stripping comments from the program, renaming variables, or partitioning the content of a single variable into two parts that are stored separately. Most of these mechanisms, however, are relatively straightforward to reverse by static analysis. In [21], opaque predicates were introduced, which are boolean expressions whose truth value is known during obfuscation time but difficult to determine statically. The idea of opaque predicates has been extended in this paper to hide constants, the basic primitive on which our obfuscation transformations rely. The one-way translation process introduced in [107, 108] is related to our work as it attempts to obscure control flow information by converting direct jumps and calls into corresponding indirect variants. The difference is the way control flow obfuscation is realized and the fact that we also target data location and data usage information. An obfuscation approach that is orthogonal to the techniques outlined above is presented in [63]. Here, the authors exploit the fact that it is difficult to distinguish between code and data in `x86` binaries and attempt to attack directly the disassembly process.

We are aware of two other pieces of work that deal with program obfuscation on the binary level. In [17], the authors developed a simple, binary obfuscator to test their malware detector. This obfuscator can apply transformations such as code reordering, register renaming, and code insertion. However, based on their description, a more powerful static analyzer such as the one introduced by the same authors in [18] can undo these obfuscations. In [111], a system is proposed that supports opaque predicates in addition to code reordering and code substitution. However, the control flow information is not obscured, and data usage and location information can be extracted. Thus, even if the opaque predicate cannot

be resolved statically, a malware detector can still analyze and detect the branch that contains the operations of the malicious code.

In [8], the authors discussed the theoretical limits of program obfuscation. In particular, they prove that it is impossible to hide certain properties of particular families of functions using program obfuscation. In our work, however, we do not try to completely conceal all properties of the obfuscated code. Instead, we obfuscate the control flow between functions and the location of data elements and make it hard for static analysis to undo the process.

Binary rewriting. Besides program obfuscation, binary rewriting is the second area that is mostly related to this research. Static binary rewriting tools are systems that modify executable programs, typically with the goal of performing (post-link-time) code optimization or code instrumentation. Because these tools need to be safe (i.e., they must not perform modifications that break the code), they require relocation information to distinguish between address and non-address constants. The reason is that address constants need to be updated to reflect the results of code modifications, while non-address constants need to remain unchanged. To obtain the required relocation information, some tools only work on statically linked binaries [89], demand modifications to the compiler tool-chain [80], or require a program database (PDB) [97, 98]. Unfortunately, relocation information is not available for malicious code in the wild, thus, our approach sacrifices safety to be able to handle binaries for which no information is present.

Besides those tools that require relocation information, a few systems have been proposed that can process binary programs without relying on additional program information. One such system is EEL [62], which can be used to optimize and instrument binaries without any program information. Another system is the binary translator UQBT [19], which is capable of translating an executable from running on one architecture to another one. Both EEL and UQBT employ a recursive disassembler and a number of heuristics to analyze binary code. However, these systems operate on RISC binaries, which is a significantly simpler task than working on the complex `x86` instruction set (especially with Windows binaries). This problem is alluded to in [85], where the authors present Etch, a system that claims to operate directly on Windows `x86` binaries. While the paper clearly

acknowledges the problem of code discovery in `x86` executables, no explanation is offered on how it is solved.

Finally, binary rewriting has already been introduced by malicious code as a means to evade detection by virus scanners. The infamous Mistfall engine [114] is capable of relocating instructions of a program that is to be infected. The idea is to create holes in the victim program so that virus instructions can be interwoven with original instructions. In this fashion, the virus code is blended with the infected program, making detection much more difficult. Interestingly, the author of the Mistfall engine states that his binary rewriting algorithm fails to correctly patch the code for jump tables that are very common in windows binaries. In our implementation, we use a heuristic that locates such jump tables and patches them accordingly after the relocation took place which allows our approach to correctly rewrite many binaries for which the Mistfall algorithm produces incorrect code.

Finding Rogue Networks A large number of studies have examined various, individual aspects of malicious activity on the Internet. In the following paragraphs, we attempt to give an overview, selecting a few representative examples for each area.

Malicious code and the Internet. Since malware represents one of the most significant threats on the Internet nowadays, it has received significant attention. Fast-spreading worms were among the the first malware subjects that were studied (for example, CodeRed [67] or Slammer [66]). Later, the focus shifted to bots and botnets [30], and a number of papers have analyzed the size of botnets [25, 40], the propagation of bots based on time zones [32], and their general behavior [82]. Besides bots, a crawler-based study explored the amount of spyware on the Internet [72]. More recently, authors also provided an overview of web-based malware [79] and the prevalence of malicious requests of search worms [93].

Scam infrastructure. Malicious code is only one facet of the flourishing underground economy, driven by criminals who aim to make quick profits. A popular way to compromise users' machines with malware is by luring them on malicious web sites that perform drive-by exploits, a phenomenon that was quantified in a

recent paper [78]. In addition to web pages, unsolicited mail is frequently used to approach victims. The spam problem was studied in [83], while the scam infrastructure that lies behind the links in emails was explored in [1]. Moreover, a number of papers examine the phishing problem (e.g., the lifetime of phishing pages [69] or the modus operandi of phishers [64]). Finally, a first attempt at studying the mechanism of the underground economy was presented in [39].

Network security. Internet-wide studies have also looked at security problems that are not directly related to malicious networks and cyber-criminals. For example, Yegneswaran et al. [112] have analyzed the characteristics and prevalence of intrusion attempts. In [68], the authors inferred the denial of service activity on a global scale by examining backscatter traffic. Finally, the subversion of the domain name service (DNS) and the rise of a malicious resolution authority was the main topic in [31].

Network reputation. The goal of previously-discussed, related work is to quantify and to deepen the understanding of specific security problems on the Internet. This is different from our work as we aim to identify networks and autonomous systems that act maliciously, without assuming that these networks have to be involved in any particular (or all possible) activities. Of course, our work is based on the observation of certain malicious activities (such as botnet command and control, spam, phishing, and drive-by-download web pages) that have been explored previously. However, we process and combine this raw data to infer those networks that are likely under criminal control.

The work closest to ours are efforts that attempt to assign a reputation to networks or an individual IP address. In its simplest form, these efforts produce blacklists of IPs that have been observed to perform malicious actions. Most often, such blacklists are used to filter spam mails [94, 96], but there are also blacklists that warn users when they visit potentially harmful web pages [41, 77]. Many of the sites that offer blacklists also compile statistics of the worst offenders, typically by counting the number of incidents in a network. Unfortunately, this technique does not distinguish between compromised, bot-infected machines and hosts in networks that are deliberately malicious. As a result, the worst offenders

are typically large networks with many customers. The goal of our work, on the other hand, is to discard the large amounts of compromised machines and identify those (often smaller) networks likely controlled by determined adversaries.

We are aware of two recent papers [16, 23] that look at temporal and spatial properties of attack sources. In [16], the authors study the spatial-temporal characteristics of malicious sources on the Internet, using data from the `DShield.org` project. The conclusion is that 20% of all IPs are responsible for 80% of the observed attacks. In [23], the authors attempt to find IPs that are clustered (spatial uncleanliness) and persistent (temporal uncleanliness) in sending spam mails, launching network scans, and hosting phishing pages. This work is closest to ours in that the behavior of hosts is used to identify "unclean" (infected) netblocks. The difference to our approach is twofold: First, we attempt to identify networks that are operated by criminals, while their work was focusing on finding bot infections. As a result, the selection of the input data sets (we include drive-by download providers and botnet C&C servers, but do not consider scanning) and the filtering techniques are different. Moreover, we combine results from multiple feeds. Such correlation efforts were not part of the previous paper.

Chapter 7

Conclusions

With the increasing number of malicious programs that are found every single day, the requirements for automatic malware analysis programs are rising steadily. In this thesis, we present novel methods that improve automatic analysis and, additionally, we show an approach to expose rogue networks.

As a theoretical result, we have shown that dynamic analysis methods are better suited for analyzing malicious programs that contain anti-analysis methods. This result gave the motivation to concentrate on automatic dynamic analysis tools in the remaining parts of this work.

To enhance current dynamic analysis methods, we have further presented a novel method that can be used to analyze binaries that try to hide their malicious behavior. Using this approach, hidden payloads can be revealed in malicious binaries and the conditions that trigger those payloads can be determined.

Finally, we have shown how it is possible to find and expose malicious networks that are used to distribute malicious software. We have presented various data acquisition and filtering methods that can accurately identify malicious networks and show those results on our web site `maliciousnetworks.org`.

Bibliography

[1] D. Anderson, C. Fleizach, S. Savage, and G. Voelker. Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In *Usenix Security Symposium*, 2007.

[2] Anubis. Analysis of unknown binaries. `http://anubis.iseclab.org`.

[3] J. Armin, G. Bruen, G. Feezel, P. Ferguson, M. Jonkman, and J. McQuaid. McColo - Cyber Crime USA. `http://hostexploit.com/downloads/Hostexploit%20Cyber%20Crime%20USA%20v%202.0%201108.pdf`, 2008.

[4] J. Armin, P. Ferguson, G. Bruen, G. Feezel, M. Jonkman, and J. McQuaid. Mccolo - cyber crime usa supplement. `http://hostexploit.com/downloads/Hostexploit_McColo_supplement_111808.pdf`, 2008.

[5] J. Armin, J. McQuaid, and M. Jonkman. Atrivo - Cyber Crime USA. `http://hostexploit.com/downloads/Atrivowhitepaper082808ac.pdf`, 2008.

[6] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The Nepenthes Platform: An Efficient Approach To Collect Malware. In *Recent Advances in Intrusion Detection (RAID)*, 2006.

[7] R. Bagnara, E. Ricci, E. Zaffanella, and P. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *9th International Symposium on Static Analysis*, 2002.

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

[9] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[10] F. Bellard. Qemu, a Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference*, 2005.

[11] D. Bizeul. Russian Business Network Study. `http://www.bizeul.org/files/RBN_study.pdf`, 2007.

[12] K. Borders, X. Zhao, and A. Prakash. Siren: Catching Evasive Malware (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[13] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Book chapter in "Botnet Analysis and Defense", Editors Wenke Lee et. al.*, 2008.

[14] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, 2006.

[15] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In *Conference on Computer and Communication Security*, 2006.

[16] Z. Chen, C. Ji, and P. Barford. Spatial Temporal Characteristics of Internet Malicious Sources. In *Infocomm Mini-Conference*, 2008.

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

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

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

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

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

[22] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical report, The University of Auckland, 1999.

[23] M. Collins, T. Shimeall, S. Faber, J. Janies, R. Weaver, and M. D. Shon. Using Uncleanliness to Predict Future Botnet Addresses. In *ACM Internet Measurement Conference (IMC)*, 2007.

[24] Computer Economics. Malware Report 2005: The Impact of Malicious Code Attacks. `http://www.computereconomics.com/article.cfm?id=1090`, 2006.

[25] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, 2005.

[26] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *International Conference on Software Engineering (ICSE)*, 2000.

[27] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[28] J. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, 2004.

[29] J. Crandall, G. Wassermann, D. Oliveira, Z. Su, F. Wu, and F. Chong. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. In *Conference on Architectural Support for Programming Languages and OS*, 2006.

[30] D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[31] D. Dagon, N. Provos, C. Lee, and W. Lee. Corrupted DNS Resolution Paths: The Rise of a Malicious Resolution Authority. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[32] D. Dagon, C. Zou, and W. Lee. Modeling Botnet Propagation Using Time Zones. In *Network and Distributed System Security Symposium (NDSS)*, 2006.

[33] D. Danchev. The Malicious ISPs You Rarely See in Any Report. `http://ddanchev.blogspot.com/2008/06/malicious-isps-you-rarely-see-in-any.html`, 2008.

[34] D. Danchev. GazTransitStroy/GazTranZitStroy Rubbing Shoulders with Petersburg Internet Network LLC. `http://ddanchev.blogspot.com/2009/06/gaztransitstroygaztranzitstroy-rubbing.html`, 2009.

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

[36] dn1nj4. The Shadowserver Foundation: RBN "Rizing". `http://www.shadowserver.org/wiki/uploads/Information/RBN_Rizing.pdf`, 2008.

[37] Emerging Threats. `http://www.emergingthreats.net/`.

[38] F-Secure Blog. Mass SQL Injcetion. `http://www.f-secure.com/weblog/archives/00001427.html`, 2008.

[39] J. Franklin, V. Paxson, A. Perrig, and S. Savage. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *ACM Conference on Computer and Communication Security (CCS)*, 2007.

[40] F. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symposium On Research In Computer Security (ESORICS)*, 2005.

[41] D. Glosser. DNS-BH - Malware Domain Blocklist. `http://malwaredomains.com/`, 2008.

[42] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, 2005.

[43] D. Goodin. 40,000 sites hit by PC-pwning hack attack. `http://www.theregister.co.uk/2009/06/02/beladen_mass_website_infection/`, 2009.

[44] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ACM Symposium on Software Testing and Analysis*, 1998.

[45] N. Gupta, A. Mathur, and M. Soffa. Automated test data generation using an iterative relaxation method. In *Symposium on Foundations of Software Engineering (FSE)*, 1998.

[46] V. Hanna. Spamhaus: Cybercrime's U.S. Hosts. `http://www.spamhaus.org/news.lasso?article=636`, 2008.

[47] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *10th SPIN Workshop*, 2003.

[48] T. Holz, C. Gorecki, F. Freiling, and K. Rieck. Detection and Mitigation of Fast-Flux Service Networks. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[49] G. Holzmann. The model checker spin. *Software Engineering*, 23(5), 1997.

[50] Honeynet Project. Know Your Enemy: Fast-Flux Service Networks. http://www.honeynet.org/papers/ff/fast-flux.html, 2007.

[51] B. Huffaker. CAIDA: AS ranking. http://as-rank.caida.org/, 2008.

[52] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *3rd USENIX Windows NT Symposium*, 1999.

[53] G. Inc. http://google.com/safebrowsing/diagnostic?site=AS:27715, 2009.

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

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

[56] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[57] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based Spyware Detection. In *Usenix Security Symposium*, 2006.

[58] B. Krebs. Taking on the Russian Business Network. http://voices.washingtonpost.com/securityfix/2007/10/taking_on_the_russian_business.html, 2007.

[59] B. Krebs. Report Slams U.S. Host as Major Source of Badware. http://voices.washingtonpost.com/securityfix/2008/08/report_slams_us_host_as_major.html, 2008.

[60] B. Krebs. FTC Sues, Shuts Down N. Calif. Web Hosting Firm. http://voices.washingtonpost.com/securityfix/2009/06/ftc_sues_shuts_down_n_calif_we.html, 2009.

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

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

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

[64] D. McGrath and M. Gupta. Behind Phishing: An Examination of Phisher Modi Operandi. In *Workshop on Large-scale Exploits and Emerging Threats (LEET)*, 2008.

[65] Microsoft Corporation. Portable Executable and Common Object File Format Specification. Technical report, 1999.

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

[67] D. Moore, C. Shannon, and kc Claffy. Code-Red: A case study on the spread and victims of an Internet worm. In *ACM Internet Measurement Workshop*, 2002.

[68] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Usenix Security Symposium*, 2001.

[69] T. Moore and R. Clayton. An Empirical Analysis of the Current State of Phishing Attack and Defence. In *Workshop on the Economics of Information Security*, 2007.

[70] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: IEEE Symposium on Security and Privacy*, volume 0, pages 231–245. IEEE Computer Society, 2007.

[71] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, 2007.

[72] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware on the Web. In *Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[73] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[74] E. Nightingale, P. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *20th Symposium on Operating Systems Principles (SOSP)*, 2005.

[75] Norman. Normal Sandbox. `http://sandbox.norman.no/`, 2006.

[76] M. Oberhumer and L. Molnar. UPX: Ultimate Packer for eXecutables. `http://upx.sourceforge.net/`, 2004.

[77] PhishTank. Clearinghouse for phishing data on the Internet. `http://www.phishtank.com`, 2008.

[78] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFrames Point to Us. In *Usenix Security Symposium*, 2008.

[79] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In The Browser: Analysis of Web-based Malware. In *Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.

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

[81] M. Rajab, F. Monrose, and A. Terzis. Fast and Evasive Attacks: Highlighting the Challenges Ahead. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[82] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *ACM Internet Measurement Conference (IMC)*, 2006.

[83] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *ACM SIGCOMM*, 2006.

[84] J. Robin and C. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Usenix Annual Technical Conference*, 2000.

[85] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Usenix Windows NT Workshop*, 1997.

[86] M. Russinovich and B. Cogswell. Freeware Sysinternals. `http://www.sysinternals.com/`, 2006.

[87] J. Rutkowska. Red Pill... Or How To Detect VMM Using (Almost) One CPU Instruction. `http://invisiblethings.org/papers/redpill.html`, 2006.

[88] B. Schneier. Gathering 'storm' superworm poses grave threat to pc nets. `http://www.wired.com/politics/security/commentary/securitymatters/2007/10/securitymatters_1004`, 2007.

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

[90] C. Seifert. Capture-HPC - Honeypot Client. `https://projects.honeynet.org/capture-hpc`, 2008.

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

[92] Shadowserver Foundation. ASN Botnet Stats. `http://www.shadowserver.org/wiki/pmwiki.php/Stats/ASN`, 2009.

[93] S. Small, J. Mason, F. Monrose, N. Provos, and A. Stubblefield. To Catch a Predator: A Natural Language Approach for Eliciting Malicious Payloads. In *Usenix Security Symposium*, 2008.

[94] SpamCop. Blocking List. `http://www.spamcop.net/bl.shtml`, 2008.

[95] Spamhaus. The spamhaus project. `http://www.spamhaus.org/`.

[96] Spamhaus. Zen: Comprehensive DNSBL. `http://www.spamhaus.org/zen/`, 2008.

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

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

[99] B. Stone-Gross, A. Moser, C. Kruegel, K. Almaroth, and E. Kirda. FIRE: FInding Rogue nEtworks. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[100] Symantec. Internet Security Threat Report: Volume X. `http://www.symantec.com/enterprise/threatreport/index.jsp`, 2006.

[101] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[102] Tool Interface Standard (TIS) Committee. Executable and Linking Format (ELF), Specification Version 1.2. Technical report, 1995.

[103] United States Computer Emergency Readiness Team. Conficker worm targets microsoft windows systems. `http://www.us-cert.gov/cas/techalerts/TA09-088A.html`, 2009.

[104] A. Vasudevan and R. Yerraballi. Stealth Breakpoints. In *21st Annual Computer Security Applications Conference*, 2005.

[105] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In *IEEE Symposium on Security and Privacy*, 2006.

[106] VMware: Server and Desktop Virtualization. `http://www.vmware.com/`, 2006.

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

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

[109] Wepawet. `http://wepawet.iseclab.org/`, 2009.

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

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

[112] V. Yegneswaran, P. Barford, and J. Ullrich. Internet Intrusions: Global Characteristics and Prevalence. In *ACM SIGMETRICS*, 2003.

[113] T. Yetiser. Polymorphic Viruses - Implementation, Detection, and Protection. `http://vx.netlux.org/lib/ayt01.html`, 1993.

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

[115] ZeuSTracker. https://zeustracker.abuse.ch/statistic.php, 2009.

Lebenslauf

12. September 1979	Geboren in St.Veit/Glan, Österreich
1990 - 1998	Bundesrealgymnasium in Feldkirchen/Kärnten Matura mit Auszeichnung
10/1999 - 05/2005	Diplomstudium Informatik an der TU Wien
05/2005	Verleihung des akademischen Grades Dipl. Ing. mit Auszeichnung Diplomarbeit “Finding Provably Optimal Solutions for the (Prize Collecting) Steiner Tree Problem” Betreuer: Prof. Dr. Petra Mutzel
11/2005 - 12/2009	Doktorratsstudium am International Secure Systems Lab der TU Wien Betreuer: Privatdozent Dr. Christopher Kruegel