ANALYSIS AND DEFENSE OF EMERGING MALWARE ATTACKS

A Dissertation

by

ZHAOYAN XU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Guofei Gu
Committee Members,	Jyh-Charn Liu
	Riccardo Bettati
	Weiping Shi
Head of Department,	Nancy Amato

August 2014

Major Subject: Computer Engineering

Copyright 2014 Zhaoyan Xu

ABSTRACT

The persistent evolution of malware intrusion brings great challenges to current anti-malware industry. First, the traditional signature-based detection and prevention schemes produce outgrown signature databases for each end-host user and user has to install the AV tool and tolerate consuming huge amount of resources for pair-wise matching. At the other side of malware analysis, the emerging malware can detect its running environment and determine whether it should infect the host or not. Hence, traditional dynamic malware analysis can no longer find the desired malicious logic if the targeted environment cannot be extracted in advance. Both these two problems uncover that current malware defense schemes are too passive and reactive to fulfill the task.

The goal of this research is to develop new analysis and protection schemes for the emerging malware threats. Firstly, this dissertation performs a detailed study on recent targeted malware attacks. Based on the study, we develop a new technique to perform effectively and efficiently targeted malware analysis. Second, this dissertation studies a new trend of massive malware intrusion and proposes a new protection scheme to proactively defend malware attack. Lastly, our focus is new P2P malware. We propose a new scheme, which is named as informed active probing, for large-scale P2P malware analysis and detection. In further, our internet-wide evaluation shows our active probing scheme can successfully detect malicious P2P malware and its corresponding malicious servers.

ACKNOWLEDGEMENTS

This is to thank my advisor, Dr. Guofei Gu, for his continuous encouragement and guidance through my PhD research. Meanwhile, I have to thank all my committee members, Dr. Liu, Dr. Bettati and Dr. Shi. Without their advising, patience, and support, it is impossible for me to finish my research and this dissertation.

Also, I would express my thanks to all my collaborators and colleague members, Dr. Christopher Kruegel, Dr. Juan Caballero, Dr. Zhiqiang Lin, Antonio Nappa, Jialong Zhang and Linfeng Chen. Whether it be a nudge at the right time, or long-term support, each contribution is important to me.

Lastly, I would like to thank my parents and friends. It is their love that supports me throughout my entire PhD study.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES viii

LIST OF TABLES x

1. INTRODUCTION 1
   - 1.1 Introduction 1
     - 1.1.1 Overcome Analysis Obstacle of Targeted Malware 3
     - 1.1.2 Proactively Protect Host from Malware Intrusion 4
     - 1.1.3 Decompose Malware's Peer-to-Peer Communication 5
   - 1.2 Overview of Solutions 6
   - 1.3 Summary of Contribution 9

2. BACKGROUND, TERMINOLOGY AND TOOLKIT 13
   - 2.1 Dynamic Analysis vs Static Analysis 13
   - 2.2 Terminology of Dynamic Malware Analysis 15
     - 2.2.1 Path Exploration in Dynamic Analysis 15
     - 2.2.2 Environment for Malware Analysis 16
     - 2.2.3 Taint Analysis for Data Flow Tracking 17
     - 2.2.4 Program Slicing for Control Flow Analysis 18
   - 2.3 Our Dynamic Malware Analysis Framework 18
     - 2.3.1 Dynamic Analysis based on Virtual Machine Introspection . . 18
     - 2.3.2 Informed Enforced Execution 19
     - 2.3.3 Flow Tracking using Taint Analysis and Program Slicing . . . 21

3. ANALYZING TARGETED MALWARE ATTACKS 22
   - 3.1 Introduction 22
   - 3.2 System Design 24
     - 3.2.1 Phase I: Pre-selection of Malware Corpus 24
     - 3.2.2 Phase II: Dynamic Environment Analysis 26
     - 3.2.3 Phase III: Distribution Deployment of GoldenEye 31
   - 3.3 Evaluation 31
     - 3.3.1 Experiment Dataset 31

3.3.2 Experiment Setup . . . 32
3.3.3 Experiment on General Malware Corpus . . . 32
3.3.4 Experiment on Known Targeted Malware Dataset . . . 36
3.3.5 Case Studies . . . 38
3.3.6 Experiment on Distributed Deployment of GoldenEye . . . 40
3.4 Limitation . . . 42
3.4.1 Correctness of Path Selection/Prediction . . . 42
3.4.2 Possible Problems of Taint Analysis . . . 42
3.4.3 Evasion through Misleading the Analysis . . . 43
3.5 Related Works . . . 44
3.6 Summary . . . 44
4. ANALYZING AND EXTRACTING MALWARE VACCINES FOR HOST PROTECTION . . . 46
4.1 Introduction . . . 46
4.2 Problem Statement . . . 47
4.3 System Design . . . 50
4.3.1 Phase I: Candidate Selection . . . 50
4.3.2 Phase II: Vaccine Generation . . . 52
4.3.3 Phase III: Vaccine Delivery and Deployment . . . 58
4.4 Evaluation . . . 59
4.4.1 Experiment Dataset . . . 60
4.4.2 Experiment on Vaccine Candidate Selection . . . 61
4.4.3 Experiment on Vaccine Generation . . . 62
4.4.4 Case Studies . . . 64
4.4.5 Experiment on Vaccine Effectiveness . . . 65
4.5 Limitation . . . 68
4.5.1 Complementary to Existing Malware Detection System . . . 68
4.5.2 Possible Evasions . . . 69
4.6 Related Works . . . 70
4.7 Summary . . . 70
5. ANALYZING AND DETECTING P2P MALWARE . . . 71
5.1 Introduction . . . 71
5.2 Problem Statement . . . 73
5.2.1 Assumption . . . 74
5.3 System Design . . . 75
5.3.1 Phase I: Malware Birthmark Extraction . . . 75
5.3.2 Phase II: MCB-assisted Network Probing . . . 79
5.4 Evaluation . . . 79
5.4.1 Experiment Dataset . . . 79
5.4.2 Experiment on Effectiveness of Portprint Extraction . . . 80
5.4.3 Experiment on Effectiveness of ICE . . . 82
5.4.4 Experiment on MCB Extraction . . . 84

5.4.5 Experiment on Detection Results through Probing 87
5.5 Limitation 90
5.6 Related Works 91
5.6.1 Multiple-path Exploration 91
5.6.2 Protocol Reverse Engineering 91
5.7 Summary 92

6. DETECTING INTERNET-WIDE MALICIOUS SERVERS 93

6.1 Introduction 93
6.2 Problem Statement 95
6.2.1 Advantages of Binary-Based Fingerprint Generation 96
6.2.2 Problem Definition 100
6.3 System Desgin 102
6.3.1 Phase I: Malware Execution and Monitoring 102
6.3.2 Phase II: Probe Generation 103
6.3.3 Phase III: Classification Function Construction 107
6.3.4 Phase IV: Probing 111
6.4 Evaluation 112
6.4.1 Experiment Dataset 112
6.4.2 Experiment Setup 113
6.4.3 Experiment on Probe Generation 113
6.4.4 Experiment on Classification 114
6.4.5 Case Studies 116
6.4.6 Experiment Setup for Scanning 119
6.4.7 Experiment on Localized Scanning 119
6.4.8 Experiment on Internet-wide Scanning 121
6.5 Limitation 124
6.5.1 Responses Check 124
6.5.2 Classification Function through Code Reuse 124
6.5.3 Fuzzing 125
6.5.4 Possible False Positive and False Negative 125
6.6 Related Works 126
6.7 Summary 127

7. LESSONS LEARNED FROM NEW MALWARE ATTACKS 128

7.1 Summary of Our Malware Analysis System 128
7.1.1 Design Goal, Guideline and Coverage 129
7.1.2 Technique Design 129
7.1.3 Performance 130
7.2 Lessons Learned 130
7.2.1 Applying Delicate Analysis in Large-scale Malware Analysis is Feasible 131
7.2.2 Revisiting of Vaccination Idea is Worthwhile 131

7.2.3 Combing Network-based Detection with Host-based Analysis is Promising 132
8. CONCLUSION AND FUTURE WORK 133
8.1 Conclusion 133
8.2 Future Work 135
REFERENCES 136

LIST OF FIGURES

FIGURE Page

1.1 Tradition of Malware Analysis and Our System Overview 6

1.2 Malware Research Framework 9

3.1 System Overview of GoldenEye 24

3.2 Working Example of GoldenEye 27

3.3 Relative Increased of Native APIs 33

3.4 Analysis Time Comparison 36

3.5 Measurement of Distributed GoldenEye 41

4.1 System Architecture of AutoVac 50

4.2 Sample Malware Code and the Traced Behavior 58

4.3 Statistics on Malware's Resource Sensitive Behaviors 62

4.4 Distribution of BDR 66

5.1 Our Two-phase Approach of PeerPress 75

5.2 Performance Comparison of ICE and Random Exploration 83

6.1 Two Network Requests Produced by Win32/Farfli.C 98

6.2 Request Generation Logic of Win32/Farfli.C 99

6.3 System Architecture of Autoprobe 101

6.4 Classification Function Example 102

6.5 Probe Generation Architecture 103

6.6 Network Request Generation logic of Win32/LoadMoney.AF. 104

6.7 Classification Function Construction 108

6.8 Probing Procedure of Autoprobe 111

6.9 Probe for Batimal Trojan 116
6.10 Probe for Taidoor Trojan 118

LIST OF TABLES

TABLE Page

3.1 Malware's Classification from VirusTotal 32

3.2 Performance comparison with two representative existing approaches 34

3.3 Test on Targeted Malware 37

4.1 Labeling Examples for OpenMutex/ReadFile 53

4.2 Malware's Classification from VirusTotal 60

4.3 Vaccine Samples (Operation Type Symbols - check Existence (E), Create (C), Read (R) and Write (W), Impact Symbol - Termination (T), Process Hijacking (H), Persistence (P), Kernel Injection (K) and Network Massive Attack (N)) 63

4.4 Evaluation on Vaccine Generation 63

4.5 Vaccine Statistics on Different Malware Families 64

4.6 Example of a High-profile Malware Vaccine 67

4.7 Vaccine Effectiveness Evaluation on Malware Variants 67

5.1 12 Malware Families in our Evaluation 80

5.2 Portprint Details of Different Malware Families 81

5.3 Running Time of MCB Extraction 84

5.4 Statistics on Extracted Malware MCBs. (Here X/Y in Column # *MCB* means there are X candidate MCBs and Y final MCBs after verification.) 85

6.1 Probe Generation Results 113

6.2 Efficiency of Classification Functions (time measured when handling 1000 continuous responses). CP: number of equation comparisons, WC: Worst Case, BC: Best Case 115

6.3 Localized Scanning Results of AUTOPROBE. #: Number of Scanners 120

6.4 Horizontal Scanning Results 121

6.5 Result of Internet-side Scanning. Here CP-x denotes CYBERPROBE and AP-x denotes AUTOPROBE. 123

6.6 Additional 3 Scanning Results of AUTOPROBE for `NoResponse` Cases 123

7.1 Summary of Our Malware Analysis Systems 128

1. INTRODUCTION

1.1 Introduction

Malicious software, also known as *Malware*, is software used or created by cyber-attackers to disrupt computer systems, gather sensitive information, or gain access to private computer systems. Typically, malware includes computer viruses, worms, trojan horses, spyware, adware, and other malicious programs. One representative example, *botnets*, which are commonly referred as collections of internet-connected computers whose security defense have been breached and control ceded to a 3rd party, penetrate millions of computers all over world every year.

The recent activity of malware attacks reflect new trend in malware's evolution. According to the Symantec Security Intelligence Report in 2013 [22], instead of attacking millions of machines on internet, well-designed malware which target at small number of specific devices, such as machines of government organizations and financial industries, has overwhelmed the field and cost incalculable economic loss every year.

The core of this new wave exists in the new functionality which has been integrated into malware itself. We observe three main evolutions in the design of malware:

- First, malware has been outfitted with the ability to detect its infection target. More specifically, such malware, which is commonly referred as *targeted malware*, can effectively collect information about the infected machine. After determining whether the machine is its target or not, it can correspondingly change its behaviors, i.e, starting infection for target machines or stopping infection for others. We refer to this as *Target Evolution*.

- Second, malware is equipped with more advanced anti-detection and anti-analysis logic. One example is that current malware may inject an *Infection Marker* onto the system to prevent duplicate infection in the same host. Such *Technical Evolution* undoubtedly has started a new round of arms race between attack and defense.

- Third, each instance of malware has been organized into larger groups with sophisticated communication structure, such as peer-to-peer structure. Different from previous IRC-based malware, which use a center server to coordinate communication, peer-to-peer structure is so flexible and covert that traditional network-based detection does not perform well to detect such malware. Because of that, it poses a great challenge to defeat such *Communication Evolution*.

The persistent evolution of malware brings great challenges to the anti-malware industry. For both malware detection and analysis, current security practitioners have been overwhelmed with traditional defending schemes. For example, at the side of malware detection and prevention, the traditional signature-based detection and prevention schemes produce ever-increasing signature databases for each end-host user, and users have to install the AV tool and tolerate consuming huge amount of resources for the required pair-wise matching between signature and running process. At the other side of malware analysis, more and more malware can detect its running environment and determine whether it should infect the host or not. Hence, traditional dynamic malware analysis can no longer find the desired malicious logic if the targeted environment cannot be extracted in advance.

Both these two problems uncover that current malware defense schemes are too passive and reactive to fulfil the task. For signature-based malware detection, since the generation and distribution of malware signature normally takes amount of pro-

cessing time, we have to passively wait until we have obtain stable signature database. For malware analysis, since there is no existing technique proactively determine whether malware is targeted at specific environment or not, we have no idea whether our analysis approach is appropriate.

In this dissertation, we conduct a series of research studies on all three trends of malware's evolution listed above. For each trend, we study the principle, feature, strength, weakness and corresponding defense and protection scheme. More specially, we propose some new techniques for the following three research topics.

1.1.1 Overcome Analysis Obstacle of Targeted Malware

The target evolution essentially changes malware's behaviors on *targeted* and *non-targeted environments/hosts*. Here, the *environment* of each host is defined as the combination of *resources*, such as file, system registry and system objects, *configuration*, such as language, key board layout, IP/MAC addresses, and *user customized data*, such as user profile and credentials. Increasingly, such environment data is used to guideline malware's behaviors. One notorious example is the Stuxnet[95] malware, which has been observed to target the Iran's nuclear infrastructure. Stuxnet infects a host when the malware determines that the host environment matches that typically present in machines that control nuclear facilities.

One important finding from the illustration is that these targeted malware has commonly adopt environment-detection logic to assist targets locating. Existing dynamic malware schemes cannot crack this logic because they cannot provide the captured malware the *desired* environment for analysis in advance.

Our proposed approach to overcome this obstacle is to dynamically change the analysis environment through speculative malware execution. In particular, we pre-capture the malware's behaviors in different environment settings. Then we apply

our new technique, speculative execution, to run analysis in multiple possible environments. Based on malware's behaviors in each environment, we dynamically adjust current analysis settings. In the evaluation, we demonstrate such design can efficiently analyze malware's targted behaviors. The detail is introduced in Chapter 3.

1.1.2 Proactively Protect Host from Malware Intrusion

Current large-scale malware intrusion coordinates in a more advanced way than ever before. One coordination technique, the *Infection Marker* [92], which has been widely used by mainstream botnet malware. Such botnet malware commonly injects one global *marker*, such as file or system mutex, onto the infected system to indicate its infection. The reason of using *Infection Marker* is to prevent duplicate infection. It is because most botnet malware aims at large-scale propagation and unavoidably they may infect the same host by multiple times. However, duplicated infection may greatly disrupt malware's functionality. For example, two duplicate malware processes may compete for same resource and consumes more computing overhead. Such behavior is so suspicious to be detected by AV tools. As a result, malware author normally chooses to inject some system-wide global mark to prevent the situation.

Even though, the technical evolution of malware brings great hardship to anti-malware community, however, it also brings chance to develop a new *protection* scheme. In this dissertation, we take a closer look at the Infection Marker technique. The technique stimulates a new research idea: If we can inject the Infection Marker in advance, could we stop the infection of these botnet malware? It is similar to inject a vaccine onto each individual host to prevent malware intrusion.

To follow the idea, we conduct a study and propose a new malware protection

technique. With the assist of our technique, malware defenders has another alternative way to defeat large scale intrusion by infecting harmless Infection Marker in advance. Moreover, with analogy to biological vaccine, we provide a complete solution to evaluate the effectiveness and side-effect of each vaccine. The detail is presented in Chapter 4.

1.1.3 Decompose Malware's Peer-to-Peer Communication

Peer-to-peer communication structure of large botnet provides more flexible and robust coordination among the enemy army. However, such structure requires both sides of malware to open one service port for peer communication. Meanwhile, malware has to be outfit with remotely-accessible/controllable logic, which is required for providing binary downloading services to new infected machines (i.e.,egg downloading), or for easier access/control to remote attackers.

In this study, we motiviate our research by asking the following question: Is it possible to utilize enemy's strength against themselves? More specially, if we can determine the port number(s) in use and further know the access/control conversation logic through that port, can we send it some crafted request and detect it by its response of our packet?

Based on that, we proposed our idea of informed active probing to detect P2P malware*. The idea is to apply automatic malware analysis technique on malware and find out the peer request which triggers the malware's *unique* response. Such request and unique response pair can be used as the detection evidence for malware. The detailed approach is presented in Chapter 5.

Moreover, we extend our informed probing approach to detect malicious servers, including P2P and HTTP servers, on the internet. We conduct a large-scale internet-

*Noting that the detection target could be any malware which opens port for peer communication, such as some Backdoor Trojan Horse.

wide scanning and detect hundreds of malicious servers. The detailed result is shown in Chapter 6.

1.2 Overview of Solutions

In this section, we overview our research solution in this dissertation. Overall, our research covers three important topics throughout three basic phases. We illustrate them in Figure 1.1.

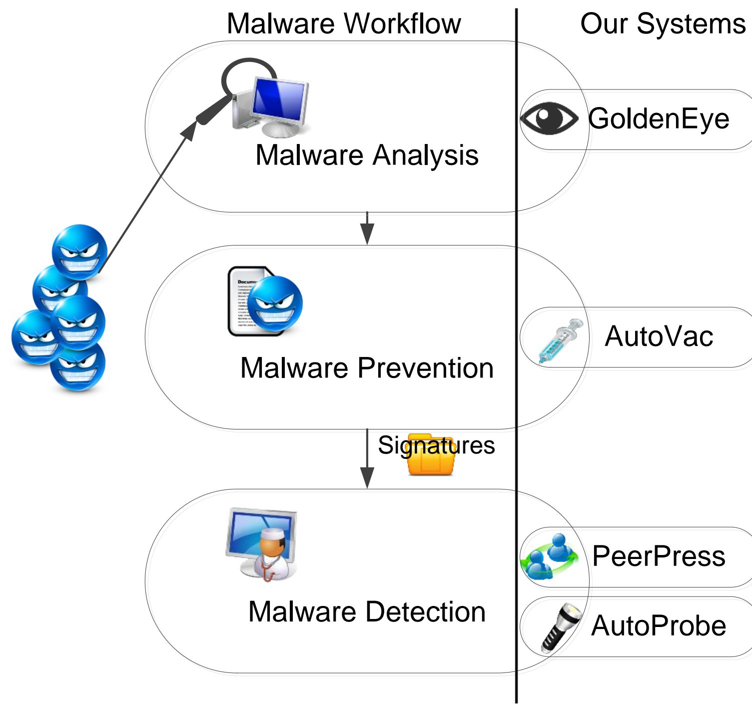

Figure 1.1: Tradition of Malware Analysis and Our System Overview

In phase I, malware researchers collect a large volume of malware corpus from users. The main research topic or challenge in this phase is how to analyze large volume of malware corpus and accurately find its malicious behaviors. Our research proposal aims to provide an effective and efficient scheme for that. As we stated,

the main reason why existing analysis schemes are not effective enough is because current schemes apply identical environment to analyze different samples. In our solution, we provide multiple analysis environments for each malware. Instead of statically constructing these environment ahead, we apply an adaptive and dynamic technique, which we called GoldenEye, to construct these environments as the same time as malware is running. GoldenEye is a dynamic program analysis technique which includes two sub-steps: *malware corpse pre-selection* and *dynamic environment analysis*. In *pre-selection*, we apply API hooking to capture all interactions between malware and its environment. From these interactions, we can deduce whether the malware possibly exhibits different behaviors in targeted and non-targeted environments. We select these *multiple-personalities* malware samples, or called targeted malware, as the input of second step. In *dynamic analysis*, we *speculatively* analyze each sample by pre-fetching the instructions of malware, emulating their execution and finding out whether some environments trigger new malicious behaviours or not. If so, we dynamically construct these environments then run the malware in all these environments. To make our analysis more efficient, we propose a distributed computing model to deploy GoldenEye in our analysis framework.

The second phase of malware research is to generate some signatures for malware prevention. Because of malware's technique evolution, it introduces a new category of malware prevention scheme, *malware vaccine*. Our second research study focuses on designing an *automatic* technique to extract the vaccines for host machines. Our proposed technique is called AutoVac which is also dynamic malware analysis technique. Specifically, AutoVac dynamically runs each malware sample and decide whether the infection mark exist or not. If so, it further determines how to use these marks as vaccines to protect our hosts. It divides into three sub-steps. In the first step, we hook the system APIs, which query the attribute of infection

marks, and test whether the return of these APIs deviates malware's logic or not. If so, we select it as a candidate. In the second step, we enforce several returns results of these APIs and collect candidate's execution trace given each enforced result. If we find one enforced result makes malware stop its malicious behavior, we can trace back and set the infection marker to generate the corresponding result. The altered infection marker is our vaccine, which can be a file, a system registry or mutex. In the third step, we emulate malware's logic and regenerate the vaccine for each of our protected end-host.

In phase III, malware researchers apply signatures to detect malware. The main research topic in this phase is how to detect malware in an effective and efficient way. As the malware's adopt new techniques for communication, the detection of these malware, such as P2P malware and the coordinated servers, become much more challenging. Our third research aims to detect these P2P malware and malicious servers. We propose PeerPress to detect P2P malware and Autoprobe to detect malicious servers. They both employ a novel technique, informed active probing, for robust and lightweight detection. Informed active probing uses port scanning, which is highly efficient to find some running server. Our informed active probing adds new features, which are extracted from malware by dynamic malware analysis, to the scanning packets. We call these new packets as *malware birthmark probings* because they can be used to differentiate valid and invalid remote responses and in further tell whether the remote server is malware's peer or not. In our internet-wide evaluation, using PeerPress and Autoprobe, we can detect hundreds of P2P malware and servers on internet. Therefore, it is a promising technique which complements existing host-based and network-based detection schemes.

In all, we combine our four systems into a generalized malware research framework, which is shown in Figure 1.2. The framework takes large volume of malware

samples as the input. GoldenEye first screen these samples and categorize them into three groups. In the first group, the samples inject some infect markers into the system and we apply AutoVac to extract malware vaccines for these malware. In the second group, the samples embed peer-to-peer or client-to-server communication logic and we employ PeerPress and Autoprobe to extract malware birthmark probings. We apply these probings to scan our network, or even internet, to detect infected machines. Overall, our framework covers malware analysis, malware signature extraction and network-level detection. We consider it can serve as a powerful framework for security researchers.

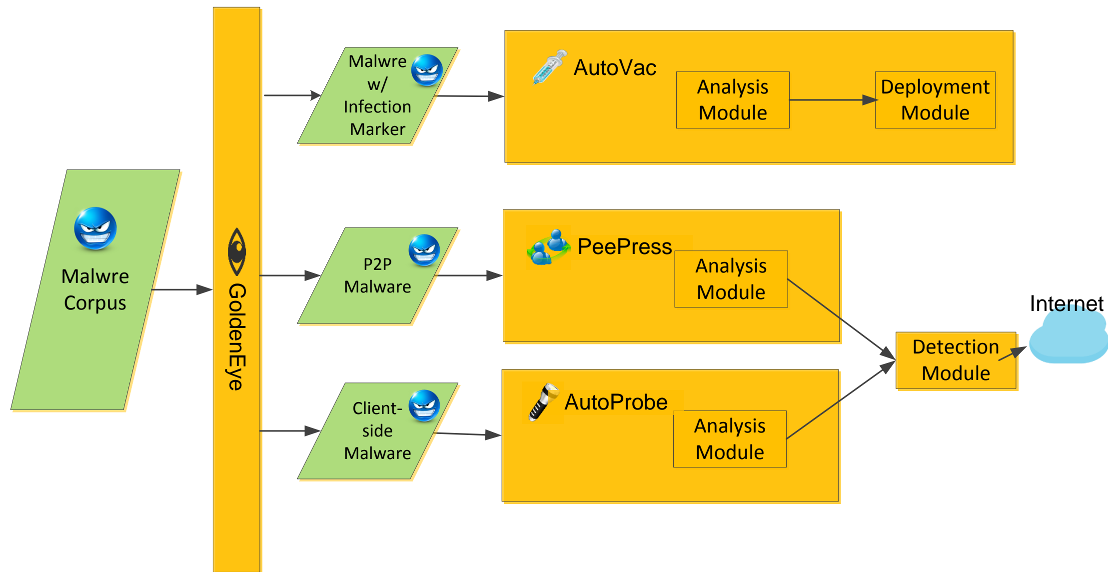

Figure 1.2: Malware Research Framework

1.3 Summary of Contribution

In this section, we summarize the contributions of this dissertation. Our workflow follows four basic steps.

In the first step, we conduct a study for each malware's evolution by analyzing large volume of malware samples. Our contribution here is to summarize the common features or behavior patterns of malware's evolution. In the second step, we propose some new techniques, such as GoldenEye, AutoVac, PeerPress and Autoprobe, to solve the new challenges brought by malware's evolution. All these techniques are the technical contributions of this dissertation. To make our techniques pragmatic, in the third step, we present our practical design of each proposed technique. We contribute by introducing multiple practice algorithms, data structures and computing models. Lastly, we evaluate our systems using large amount of real world data. From the evaluation, we provide some new findings, such as some previously unknown malicious behaviors and some new discovered malicious servers, as our contribution.

Next, we delicately explain our contribution in each research topic.

Through studying recent malware's target evolution, we provide some insights for analyzing targeted malware. We propose a new analysis system, GoldenEye, for large-volume malware analysis. As a malware analysis system, GoldenEye makes the following contributions:

- We find some common behavior patterns about targeted evolution of new malware attacks. Based on these findings, we further demonstrate why existing malware analysis schemes may fail at analyzing these malware.
- We present GoldenEye, a malware analysis tool which provides a better trade-off between effectiveness and efficiency, an important and highly demanded step beyond existing solutions.
- We design and implement a distributed version of GoldenEye to discover malware's targeted environments by applying novel speculative execution in dynamic, parallel, virtual environment spaces. The proposed approach facili-

tates conducting large volumes of malware analysis in real-time fashion.

- We provide an in-depth evaluation of GoldenEye on real-world malware corpus and show that GoldenEye successfully exposes malware's environment-sensitive behaviors with much less time or fewer resources, clearly outperforming existing approaches. We also show that GoldenEye can automatically identify and provide correct running environments for tested well-known targeted malware families. To further improve the accuracy and efficiency, we also propose a distributed deployment scheme to achieve better parallelization of our analysis.

Through studying recent malware's technical evolution, we introduce a new intrusion prevention scheme. Different from previous IPS system which requires complicated rules, our approach can simply inject a system resource, such as a empty file, to prevent malware intrusion. In particular, we make the following contributions:

- We conduct a systematic study of malware's technique evolution. Through studying the mechanism of malware's duplication prevention technique, called Infection Marker, we propose our malware vaccination idea to take advantage malware's strength against itself.
- We present the problem and challenge of malware vaccination. Based on that, we discuss all possible mutable system resources of our vaccine interest, and present a taxonomy of malware vaccines.
- We design and implement AutoVac, which can automatically extract some system resource as malware vaccine. Meanwhile, AutoVac can automatically evaluate the effectiveness of vaccine.
- We evaluate our AutoVac with a large set of real-world malware samples. Experimental results show that it is truly possible to generate working vaccines for many real-world malware families, such as Conficker, Sality, and Zeus. Thus,

we prove that we can use vaccines as a complementary approach in real world practice.

Through studying recent malware's communication evolution, we find we can use malware's communication logic, which is referred as malware birthmark, to detect them. We make the following contributions for designing new detection schemes:

- We conduct a systematic study of malware's communication evolution. We focus on analyzing malware's peer-to-peer and server-to-client communication logic and find such logic is unique enough to differentiate each malware family
- We propose the new detection strategy combining host-level dynamic malware binary analysis and network-level informed probing techniques.
- To detect peer-to-peer malware, we develop PeerPress, a prototype system that implements the proposed framework. We design new techniques to determine if given malware opens a specific port and automatically extract the port generation algorithm/logic. Meanwhile, we develop Autoprobe, a tool for automatically generating active probing fingerprints to detect remote malicious servers.
- We evaluate PeerPress and Autoprobe with multiple representative and complex real-world malware families. Moreover, we conduct internet-wide probing evaluation, and our tools successfully detect hundreds of malicious servers on Internet.

2. BACKGROUND, TERMINOLOGY AND TOOLKIT

In this dissertation, we mainly discuss malware's threat from a malware analyst's view. The main task of malware's analyst is to analyze malware sample and determine the most effective way to detect malware and protect users. Meanwhile, the efficiency of analysis itself is also the focal point because there are large volume of malware samples collected everyday. In this chapter, we discuss some background and terminology of malware analysis.

2.1 Dynamic Analysis vs Static Analysis

In the word bank of malware analyst, dynamic analysis and static analysis are two commonly-used terms. Static analysis is a set of analysis techniques which directly disassemble malware binary and analyze its logic. Static analysis is very effective because the analyst is able to obtain a complete view of malware logic. However, in practice, when we apply static disassembling on malware, we may lost the track and find that malware can not be disassembled. It is a common situation in the field. The reason why static analysis fails in front of malware is because there are some anti-disassembling code, such as junk and encoded/packed code, embedded in the malware binary. Such code can prohibit the common dissembling procedure, and without proper decoding/decryption in advance, there is no way for us to obtain the correct code of malware logic.

To further illustrate why static analysis is not applicable to automatic malware analysis, we list two anti-dissembling techniques:

- Code Packing [77]: A special infection technique which compresses or encrypts the content of host program. It is an effective way to distort static analysis.
- Junk Code [77]: Malware inserts junk code to its source and repackage itself. In

this way, the repackaged malware looks completely different from the previous one. Meanwhile, the injected junk code can also mislead the disassembling by providing unaligned instructions and data.

To overcome the limitation, dynamic analysis techniques become the natural choices for current malware analysts and researchers. Since the dynamic analysis is operated on malware's execution result, hence, the anti-disassembling techniques can no longer work to prevent analysis.

In particular, dynamic analysis is conducted directly on malware's execution result, which is referred as *execution trace*. The standard procedure of dynamic malware analysis includes three steps:

- Execution Monitoring: First, malware is executed with some monitoring facilities. Previously, process-level debugger is an ideal choice for analyst since the debugger can directly access malware's execution information, such as memory and execution sequence. However, current malware also embeds some logic to detect whether it is running with some debugger or not. If so, the malicious behaviors will be hidden. As a result, nowadays, the analyst choose to run malware in a virtual machine and construct the monitor outside of the virtual machine. The monitor provides an instruction-level execution tracking for the malware process.

- Trace Collection: With the assist of VM monitor, the next step of dynamic analysis is to collect malware's execution trace. The execution trace may include all executed instructions, register values, memory access result and system calls. As described, the execution trace is essentially one path of malware's execution logic which is triggered by some predefined analysis goal. For example, if analyst want to know the malware's execution logic of parsing network

input, she can choose only to collect the execution path which is triggered by network input. Thus, one way of viewing dynamic analysis is static analysis of straight-line code. The code is straight-line since it only includes a specific execution path. The difference between a static and dynamic approach is that we use an input to select the exact execution path in the dynamic case but in the static case, we cannot trigger specific path.

- Trace Analysis: The next step of analysis is directly performed on the execution trace. The main tasks of analysis include data-flow analysis, which tracks how malware processes the input data and how it generates its output, and control-flow analysis, which deduces what is the trigger condition of malware's behavior and determines whether there exists some alternative execution paths. The analysis result generates a detailed malware's behavior profile, which is studied to formulate final defense and protection scheme.

In this dissertation, we apply dynamic analysis on all malware binary samples to discover new features of malware attack. In next section, we introduce the terminology of dynamic malware analysis.

2.2 Terminology of Dynamic Malware Analysis

2.2.1 Path Exploration in Dynamic Analysis

The coverage of dynamic analysis is only constraint to the trace we generated for one specific execution path. Hence, triggering malware to execute the desired path is an important task for malware analysts. However, the execution of malware is dependent on many elements, such as the current system configuration and network input, and moreover, for analysts, we lack a clear definition or description about the desired malicious path before any concrete analysis task.

As a result, in practice, we take a more reactive way to conduct analysis, which is called *path exploration* in our terminology. The rationale behind path exploration is *iteratively enforcing* malware to execute some or all possible execution paths, evaluate which one is possible malicious path, and in further, figure out what is the trigger condition of the path.

To enforce malware's execution, the *forced execution* technique [93] has been widely adopted in practice. The idea is simply to enumerate all possible results for branch instruction and collect all corresponding traces. Even though it is a tedious and resource-consuming process, it still has been used in practice if the analysts can strictly define what branches they want to explore.

The task of defining the desired branch is fulfilled by another technique, *symbolic execution* [71]. The intent of conducting symbolic execution is to allow analyst define what is the focal point of their analysis. For example, analysts can define the analysis is to discover malware's handling logics for network input, and accordingly, they can treat each byte of network input as one symbol instead of a concrete value. Then symbolic execution executes the malware binary and collects all the branches which are sensitive to the symbolized value. With the assist of symbolic execution, forced execution can limit its forced execution range to a much smaller set of branches. The last task of symbolic execution is to deduce the valid range of each symbol for each execution path. It is achieved by *constraint solving* technique [68] which has been applied in program flow analysis.

2.2.2 Environment for Malware Analysis

Analyzing malware is more of an art than a technique. One difficult part of analysis is to figure out the dependence between malware's logic and its running *environment*.

For example, malware commonly employs system-wide process enumeration to find whether there are some debugging process existed along with itself or not. If so, it may choose to hide its behavior or simply kill the debugger process to escape from analysis.

Therefore, selecting a proper analysis environment for malware's execution is important, especially when our defined environment has included so many elements. In our terminology, we define the following two categories of elements as the analysis environment:

- System Resources, such as file, registry, mutex, program library, communication pipe, devices and so on.
- System Configuration, such as time, OS version, language, keyboard layout and so on.

2.2.3 Taint Analysis for Data Flow Tracking

One important existing technique to assist analyzing data flow is taint analysis [71]. The purpose of dynamic taint analysis is to track data flow between some predefined source(s) and sink(s). It assigns *taint* symbol, which is called tainted data, to each data source. Then the analysis tracks how tainted bytes are processed by each instruction. Under the context of malware analysis, we can apply it to find how malware processes the data.

One application of taint tracking in malware analysis is to figure out how malware reacts to its master commands. To find that, we can taint all the network input data and monitor the propagation of such data. Since each instruction contains source and destination operands, thus, if we find any instruction's source operand is tainted data, we accordingly taint the destination operand. Accordingly, if the tainted destination

operand is written by a *clean* source, we clear the taints of data. Such process repeats till we hit pre-defined sink point, i.e.,the launch of network scanning.

Another application is using reverse or backward taint analysis to track what is the source of malicious data. One exemplary application is tracking how malware dynamically generates its contacted domain, which is also referred as Domain Generation Algorithms(DGA). In this case, we taint the data of domain name and backward conduct taint analysis till we find some interesting system/function call, i.e.,some random data generation function.

2.2.4 Program Slicing for Control Flow Analysis

At the other side, control flow analysis is also important to understand malware's logic. The *program slicing* is one representative technique for control flow analysis. The goal of the program slice is to find all sources that indirectly influence the value of taint data. Therefore, the output of program slicing is one independent program, which is extracted directly from malware but fulfill one independent logic, such as DGA calculation. The slice program is useful when the analyst want to regenerate some data without running the complete malware program again.

2.3 Our Dynamic Malware Analysis Framework

Choosing appropriate malware analysis techniques is important to our study. As we discussed before, we select dynamic analysis as the foundation of our framework.

Meanwhile, to overcome the limitation of dynamic analysis, we present our designs and techniques as follows:

2.3.1 Dynamic Analysis based on Virtual Machine Introspection

The first important design of our analysis is using Virtual Machine Introspection, VMI, technique to implement our analysis toolkit. To be specific, our toolkit is built

upon open source project [100, 32]. The Qemu is a machine-level emulator which emulates the X86 instruction set. Based on the structure of Qemu, we could have a complete view of malware's running behaviors, which includes how malware is booted, how it accesses system resources and how it communicates with the outside connections.

2.3.2 Informed Enforced Execution

One novel design of our platform is to conduct a more effective path exploration scheme in the analysis. Instead of being tangled with complex conditions solving in symbolic execution and forced execution, we choose to use another way to explore malicious logic. We enforce malware to execute *some discovered malicious code* by some predefined guideline. We refer this proposed technique as *Informed enforCed Execution* or ICE.

The underneath heuristic of ICE is that malware employs system functions/calls as the executor of their malicious intent. For example, malware may call system function `send` to send attacking packets. Therefore, the malicious path should contain the instruction which calls `send`. ICE is targeting to make wise exploration and enforce malware to execute the path which includes these *executor functions.* To be specific, we introduce two novel exploration guidelines to efficiently identify MCB paths: (1) Enlarge the executor functions, or sinkhole function, hit range using Function Containers (2) Make wise decision on branch points by Foreseeing.

Function Containers Inside Malware. We employ directed path exploration for finding malicious paths, e.g., typically containing a sinkhole point of network transmission routines such as a send library call. In particular, to expand such limited small number of sinkholing routines to a larger hit surface, we introduce the concept of Function Containers to assist directed exploration.

Definition: *A function container is a function satisfying any of the following conditions:*

(I) Any desired sinkholing system/library calls are automatically function containers, i.e., $SysCall_{desired} \in FC$*;*

(II) The function directly or indirectly contains/wraps an existing function container. Furthermore, the call of this FC will lead to the call of $SysCall_{desired}$.

During the online path exploration, we follow a breadth-first principle and enforce the execution towards code blocks containing high priority FCs (e.g., those will lead to network transmission routines). At the same time, we also update our FC hashtable if our initial FCs collection is not correct or not complete. We use two policies to update the FC hashtable: (1) If one trace shows that after entering a certain FC the trace does not lead to the desired system/library call, we delete it and its upper level FCs from the FC list (since it violates condition II); (2) If we find one critical system call executed but not yet defined in the FC hashtable, we create a new set of level-n containers for this system call.

Note that, we can not only find some desired function containers, but also we can find some function containers which may possibly stop malicious behaviors. For example, some functions which wrap the functions such as `CloseSocket`, `ExitProcess` terminates malware's execution. When we explore malicious path, we should prevent malware from going through these functions.

Branch Foreseeing. Our enforced execution needs to make decisions at each branch point to determine which path to take/prefer. We leverage *Foreseeing* for this purpose.

In detail, we foresee (statically look forward) k code blocks to search for the calls to any recorded function container. If a high priority FC is contained in a code block,

ICE assigns a priority score of +1 for the block. Similarly, it assigns −1 in the case of encountering a low priority FC. Then, ICE simply sums up the total priority scores Λ among all code blocks in the $Left$ and $Right$ branches and gives preference to the branch with the overall higher priority score. We enforce the branch decision [93] at such branches and repeat the foreseeing till we hit a target FC. To prevent exploring the same path again, we set the code block that we have explored as low priority.

For the case that priority score $\Lambda_r = \Lambda_l$, the exploration follows the natural execution choice.

2.3.3 Flow Tracking using Taint Analysis and Program Slicing

The third challenge we need to solve is to find the code that we are interested in. To solve the problem, we conduct both data flow tracking and control flow tracking in our analysis tool.

Taint Analysis for Data Flow. For data flow tracking, we construct a taint analysis [60] component and a program slicing [51] in our toolkit. Combining both taint analysis and program slicing, our tool can efficiently find all the instructions we need for in-depth analysis. In next chapter, we start presenting our research about defending targeted malware's intrusion.

From the next chapter, we start presenting our research works about the evolution of malware attacks. Firstly, we take a closer look at the target evolution of recent malware.

3. ANALYZING TARGETED MALWARE ATTACKS

3.1 Introduction

In the past few years, we have witnessed a new evolution of malware attacks from blindly or randomly attacking all of the Internet machines to targeting only specifically designated systems, with a great deal of diversity among the victims, including government, military, education, civil society networks [23], and even Fortune 500 companies [7]. Among them, advanced persistent threats (APT), a unique category of targeted attacks that sets its goal at a particular individual or organization, are consistently increasing and they have caused persistent and enduring damage to each victim [76]. According to the annual reports from Symantec Inc, in 2011 targeted malware has a steady uptrend of over 70% increasing has been observed since 2010 [76], such overgrowth has never been slow down, especially for the growth of malware binaries involved in the targeted attack in 2012 [66].

To defeat this trend of malware attacks, we believe a compelling defense should be able to agilely expose the attack's target and in further to prevent the similar victims from being infected. Nowadays, security practitioners are working hard to identify targeted attacks through screening huge volumes of data. While current practice such as screening targeted emails could detect a potential targeted attack in every two million emails [76], we think such schemes are not efficient enough because it only limits the scope in attacks' propagation.

In contrast, since the targeted malware sample carries out the attack, proactive analysis on malware samples may be more efficient rather than passively monitoring. More specifically, if we can derive the environment condition(s) which trigger malware's malicious behavior, we can promptly send out alerts to the systems satisfying

these conditions.

To this end, we have to re-factor our existing schemes to targeted malware analysis infrastructure, especially for dynamic malware analysis. However, existing dynamic analysis techniques are not effective and efficient enough, and they have to address two additional challenges: First, it requires *highly efficient techniques* to handle a great number of targeted malware samples collected every day. Second, it requires the analysis environment be more *adaptive* to each individual sample since malware may only exhibit its malicious intent in its targeted environment.

As such, we attempt to fill the gap for more efficiently unveiling the malware's targeted environment. Specifically, we present a novel systematic dynamic analysis scheme, GoldenEye, for agile and effective malware targeted-environment analysis. To serve as an efficient tool for malware analyst, GoldenEye is able to *proactively* capture malware's environment-sensitive behaviors in progressive running, *dynamically* determine the malware's possible targeted environments, and *online* switch its system environment for further analysis.

The key idea is that by providing several dynamic, parallel, virtual environments (*not* virtual machines) during a single malware execution, GoldenEye proactively determines what the malware's targeted environment is, through a specially designed speculative execution technique to observe malware behaviors under alternative environments. Moreover, GoldenEye dynamically switches the analysis environment and let malware itself expose its target-environment-dependent behaviors. While GoldenEye trades space for speed, interestingly our experimental results show that GoldenEye can actually achieve much higher speed than existing multi-path exploration like techniques with much less memory space.

3.2 System Design

In this section, we discuss the details of our scheme, GoldenEye, to analyze the emerging attack of targeted malware. The system architecture is illstrated in Figure 3.1.

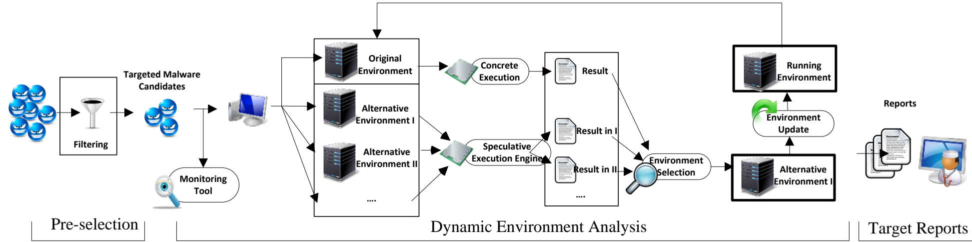

Figure 3.1: System Overview of GoldenEye

3.2.1 Phase I: Pre-selection of Malware Corpus

The first phase of GoldenEye is to quickly obtain the malware samples which are candidates of targeted malware. Our criteria for the pre-processing is to find any malware that is sensitive to its running environment.

Our scheme of pre-selection is achieved by tainting the return values of certain environment query API/instructions and tracking whether the tainted bytes affect the decision on some branch instructions, such as changing `CFlag` register. If the tested sample is sensitive to its environment querying, we keep the sample as the targeted malware candidate.

API Labeling. The most common way for malware to query its running environment is through certain system APIs/instructions. To capture malware's environment queries, we need to hook these APIs/instructions. Furthermore, it is important

to derive all possible return values of these APIs/instructions because these return values are used to define parallel environments.

In GoldenEye, we label three categories of environment queries:

- *Hook system-level environment query APIs.* The operating system provides a large set of system APIs to allow programmer query its running environment. They have also been commonly used by malware to achieve the similar goal.
- *Hook environment-related instructions.* Some X86 instructions such as `CPUID` can also be thought as a way to query environment information.
- *Hook APIs with environment-related parameter(s).* Some system files/registries can be used to store environment configuration. Thus, we also hook file/registry operation APIs and examine their parameters. If the parameters contain some keywords, such as `version`, we also treat as a query attempt.

For each labeled API/instruction, we examine its return value as the reference to initialize parallel environments. In general, we construct one speculative execution context for each possible return value. To narrow down the alternative choices of the environment, we define the following four basic sets of return values.

- *BSET(n)* defines a two-choice (binary) set. One example for `NtOpenFile` is the *BSET(0)* for return value `NTSTATUS`, which accepts 0 (success) or other value (failure).
- *SET([...])* defines a normal enumeration of values, such as enumeration for `LANGID` in the default system language.
- *RANGE(A, B)* set contains a range of possible return values.

Based on these three sets, we design dynamic ways for constructing the parallel contexts along with malware analysis. For example, we simply construct two parallel contexts for *BSET(n)* element. Note that a large amount of system objects, whose querying API return -1 as *non-existence* and *random value* as the *object handle*,

belong to this type. We consider all these objects as *BSET(n)* element.

For *SET([...])* with n different values, we accordingly initialize n parallel settings based on the context.

For *RANGE(A, B)* set, we examine whether the range set can be divided into some semantically independent sub-ranges. For example, the different range of `NtQuerySystemInformation`'s return specifies different type of the system information. For these cases, we construct one context for each semantically-independent sub-range. Otherwise, we initially construct one context for each possible value.

3.2.2 Phase II: Dynamic Environment Analysis

Dynamic environment analysis is the main component of GoldenEye. We use Conficker [63] worm's logic as a working example. As illustrated in Figure 3.2, in this example, Conficker worm queries the existence of specific `mutex` and the version of the running operating system. The malicious logic is triggered only after the check succeeds.

Initialization of Malware Environment Analysis. After the preprocessing, we first initialize the analysis by constructing parallel environments when we find malware's environment query. We define a running environment with a set of environment elements as

$$env = \{e_1, ..., e_i, ...e_n\}$$

For each e_i, it is defined as a tuple:

$$< identifier, API, type, value >$$

where $identifier$ uniquely denotes the name of each environment element, such as the `mutex` name or the title of GUI windows, API is the invoked API to query the

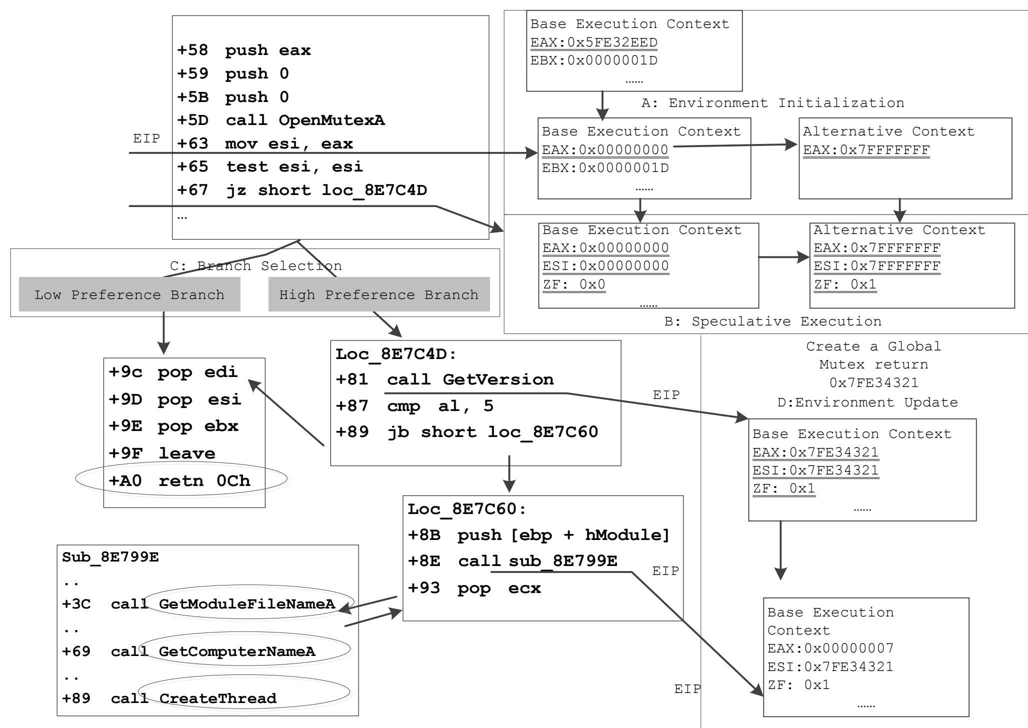

Figure 3.2: Working Example of GoldenEye

element, *type* specifies the type of element, such as system setting (system language, os version, etc.) or system objects (the existence of files, registries, etc.), and *value* states *what are possible values of each element*, such as true/false or a set of hex values.

Context Maintenance of Speculative Execution. After GoldenEye captures malware's environment query, a set of initialized environment contexts are maintained by our speculative execution engine. The main overhead of our speculative execution comes from continuously maintaining those parallel contexts.

To save space, the key design for context maintenance is based on our progressive execution scheme. Since the execution in parallel can be naturally synchronized by each instruction (it follows the same code block(s)), we choose to only record the

modification of parallel contexts. As illustrated in Figure 3.2 step A and B, we have no need to maintain the full execution context, such as all general registers value and execution stack, in each parallel space. We only track the different data, which is `EAX` and `ESI` in the example. We maintain such alternative contexts using a linked list. When an environment update operation starts, we only update the *dirty* bytes that have been modified since the previous block(s). In further, we organize each progressive context using linked-list to track the modified bytes.

Taint-assisted Speculative Execution. Another key design to prevent redundant overhead is to apply taint tracking on environment-sensitive data. In particular, we taint each byte of the environment query's return and propagate the tainted labels by each instruction. When we encounter an instruction without tainted operation, we continue with concrete execution. Otherwise, when we encounter an instruction with tainted operands, we accordingly update the execution context in all alternative environments. We continue such propagation until we reach the end of a basic code block. For the branch instruction, we also determine whether it could be affected by the tainted bytes or not (whether `CFlag` has been tainted or not). If it is an environment-sensitive branch, we continue the branch selection and environment update. If not, speculative execution starts a new pre-fetch operation to continue analyzing a new code block.

The advantage of using taint analysis is to efficiently assist the analysis in three ways: (1) Our speculative execution is only conducted on the instructions whose operands have been tainted. It allows us to skip (majority) untainted instruction for speculative execution to save analysis effort. (2) Tainted propagation can help us to determine the environment-sensitive branches. Our environment prediction/selection is based on the correct identification of these sensitive branches. (3) Tracking the

status of the tainted label helps us to maintain parallel environment spaces and delete/merge untracked environments.

Heuristics for Branch Selection. Next, we present how we evaluate the branch through the assist of informed forced execution. In GoldenEye, we apply three heuristics to determine what is a possible branch in the targeted environment:

- If a branch contains a function call that calls some exit or sleep functions, such as `ExitProcess`, `ExitThread`, and `sleep`, it means this branch may terminate the program's execution. We treat another branch as the possible targeted branch.
- If a branch contains function calls that create a new process or thread, such as `CreateProcess` and `CreateThread`, or start network communication, such as `socket` and `connect`, we treat this branch as the possible targeted branch. Similar function calls could be some representative malicious calls, such as functions for process injection, auto-booting, and kernel hijacking [44].
- If a branch directly interacts with the environment, we treat this branch as the possible targeted branch. For example, if malware creates a file before the branch, we treat the branch that directly operates on the created file as the targeted branch. Essentially, if one branch contains instructions intensively operating on tainted data, we consider it as the targeted branch.

After examining these three heuristics, if we still cannot decide the possible targeted branch in a given time window, inspired by the multi-path exploration work [57], we will save the snapshot at the branch point and conduct the concrete execution for both branches.

Environment Update. The result of target branch predication is to decide whether to remain in the current running environment or to switch to another alternative en-

vironment. If the environment switching is needed, there are three basic environment switching operations: (1) Creation, (2) Removal, (3) Substitution.

The key requirement of our design is to update the environment online. Hence, our environment update step is performed directly after the speculative execution engine has committed its execution context.

Creating an element is a common case for an environment update. Especially when malware tries to query the existence of certain system object, we would thus create such an object to ensure the following operations on this object will succeed. To this end, we create a dummy object in the system, such as creating a blank file with certain file name or creating a new registry entry. Accordingly, deleting the element is the opposite operation and we can simply achieve that by deleting the corresponding existing system object.

The substitution operation usually occurs when malware requires different system configuration from the current running environment. A main approach to find out the correct environment setting is through the result of the speculative execution. Since the speculative execution tells us the condition to ensure the selected branch, we can concretely set up the value to satisfy this requirement. For example, we can modify some registry entries to modify certain software version. As a more generic solution, we design an API manipulation scheme. When a substitution occurs, we hook the previously captured APIs or instructions, and return a manipulated value to malware for every query.

The environment update for our working example is illustrated in Figure 3.2 step D. The first step is to update the base execution context as the selected context. In the example, we first update the `ESI` and `ZF` register. Secondly, since `EAX` is the object handle of the `mutex` object, we need to create the `mutex` for current context and bind `EAX` to the `mutex` handle. In our implementation, we do not concretely create the

`mutex`. Instead, we record the handle value and when any system call operates on the handle, we enforce the `SUCCESS` to emulate the existence of the object.

3.2.3 Phase III: Distribution Deployment of GoldenEye

To further improve the accuracy and efficiency, we propose a distributed deployment scheme of GoldenEye. The scheme is essentially taking advantage of parallel environments created by the speculative engine and distributing them to a set of worker machines for further analysis.

In detail, when the speculative engine detects an environment-sensitive branch, it can choose to push a request R into a shared task queue and allow an idle worker (virtual) machine to handle the further exploration. The worker machine monitoring (WMM) tool pulls each request and updates the environment settings before analyzing a malware sample. After booting of a malware sample, the WMM tool will monitor the execution status and enable the speculative execution if some unobserved malicious logic has occurred.

3.3 Evaluation

3.3.1 Experiment Dataset

Our test dataset consists of $1,439$ malware samples, collected from multiple online malware repositories such as Anubis [3] and other sources [61]. This dataset is randomly collected without any pre-selection involved. We analyze these $1,439$ malware using a free virus classification tool [88] and classify them into 417 distinct malware families. Analyzing the classification result, we further categorize these 417 malware families into four classes: *Trojan*, *Worm*, *Spyware/Adware*, and *Downloader*. The statistics about our dataset is listed in Table 3.1. Meanwhile, we also collect a small dataset that includes some well-known targeted malware samples such as Conficker [63], Duqu [80], Sality [85], and Zeus [86]. For each malware family, we

collected several variant samples.

Category	# Malware Samples	Percent	Distinct Families
Trojan	627	43.57%	263
Adware/Spyware	284	19.73%	59
Worm/Virus	185	12.85%	27
Downloader	343	23.83%	68
Total	1,439	100%	417

Table 3.1: Malware's Classification from VirusTotal

3.3.2 Experiment Setup

In our experiment setting, we manually labeled 112 system/library APIs with 122 output parameters, and hooked them in our analysis.

All our experiments are conducted in a machine with Intel Core Duo 2.53GHz processor and 4GB memory.

3.3.3 Experiment on General Malware Corpus

We conduct the following experiments to evaluate GoldenEye on the larger malware dataset with 1,439 samples.

Measurement of Effectiveness. First, we study the effectiveness of our approach in terms of the code coverage in analysis. To measure that, we first collect a baseline trace by naturally running each malware sample in our virtual environment for 5 minutes. Then we apply GoldenEye to collect a new trace in the dynamically-changed environment(s). In our evaluation, we measure the relative increase in the number of native system calls between the base run and analysis run. The distribution of increased APIs among all malware samples is shown in Figure 3.3. As seen in Figure 3.3, over 500 malware samples exhibit over 50% more APIs in the new run. It shows that our system can expose more malware's environment-sensitive behav-

iors. From the result, we also find that over 10% Adware/Spyware exhibits 100% more behaviors. It may imply that Spyware is more sensitive to the running environment compared with other malware categories. This is reasonable because Spyware normally exhibits its malicious behavior after it collects enough information about the infected user. This further proves the usefulness of our system. Examining the quantitative results of other categories, it is evident that our system can efficiently discover malware's environment-sensitive functionalities.

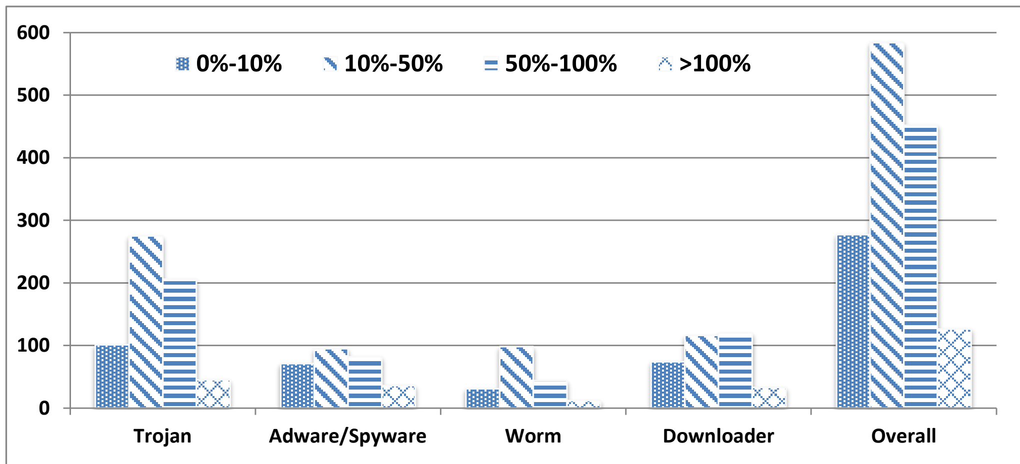

Figure 3.3: Relative Increased of Native APIs

Comparison with Related Work. The last set of our experiment is to compare the effectiveness and efficiency of GoldenEye with other approaches. To this end, we first implemented the approach presented in the related work [57] (labeled as Related Work I), which needs to explore multiple possible paths of environment-sensitive branches. Secondly, we configure four virtual environments according to the descriptions in related work [53] (labeled as Related Work II). We test malware samples in all four environments and choose the best one as the result. Then we

randomly select 100 malware samples from each category of malware and collect the traces generated by GoldenEye, Related Work I, and II, respectively. When collecting each execution path trace, we terminate the analysis if no further system calls are observed for 30 seconds (e.g., sample terminates or sleeps), or if it reaches maximum analysis time which we set as 300 seconds (5 minutes) for GoldenEye and Related Work II. For Related Work I, since it needs to explore all possible paths, we have to let it run for a much longer time. However, it could possibly take forever. Hence, in this experiment we limit its maximum analysis time to 12 hours.

Approach	Malware	Percent of Increased APIs				# of Rolling Back		
		<10%	10%-50%	50%-100%	>100%	<50	50-500	>500
GoldenEye	Trojan	31%	36%	27%	6%	74%	26%	0%
	Adware/Spyware	29%	34%	28%	9%	86%	14%	0%
	Worm	39%	47%	11%	3%	84%	16%	0%
	Downloader	43%	29%	24%	4%	69%	31%	0%
Related Work I[57]	Trojan	21%	34%	29%	16%	0%	2%	98%
	Adware/Spyware	16%	32%	33%	19%	0%	1%	99%
	Worm	27%	28%	37%	8%	0%	0%	100%
	Downloader	19%	41%	23%	17%	0%	2%	98%
Related Work II[53]	Trojan	94%	5%	1%	0%	-	-	-
	Adware/Spyware	99%	0%	1%	0%	-	-	-
	Worm	96%	4%	0%	0%	-	-	-
	Downloader	98%	2%	0%	0%	-	-	-

Table 3.2: Performance comparison with two representative existing approaches

The result is presented in Table 3.2. We use the following metrics for the comparison:

- Increased APIs. For each of three approaches, we pick the longest trace during any single run to compare with the normal run. For each approach, we record the percentage of malware samples whose increased APIs belonging to $0-10\%$, $10-50\%$, $50-100\%$, or 100% and above. From the result, we can see that Related Work I performs the best among all approaches, which is obvious

because this approach blindly explores all possible paths and we select the path with most APIs in the comparison. Meanwhile, in our test, pre-configured environment (Related Work II) can seldom expose malware's hidden behaviors; on average it only increase 5% more APIs. Thus, even though pre-configured environment has no extra overhead for the analysis, it cannot effectively analyze targeted malware. It further confirms that it is impractical to predict malware's targeted environment beforehand. Our approach clearly performs significantly better than Related Work II, and very close to Related Work I.

- Number of Rolling Backs, which is a key factor to slow down analysis. For exploring both branches, Related Work I has to roll back the execution. In theory, for each environment-sensitive branch, it requires one roll back operation. From the result, we can see that most of the samples have to roll back over 500 times to finish the analysis. However, our GoldenEye can efficiently control the number of rolling back because it only occurs when branch prediction cannot determine the right path to select. The largest number of rolling back in our test is 126 and median number is 39. It means that we can save more than 90% overhead when compared with multi-path exploration.

Finally, we also compare the total time to complete analysis for GoldenEye and Related Work I.

For each malware, both GoldenEye and Related Work I may generate multiple traces and we sum up all the time as the total time to complete the analysis of the malware. The result is summarized in Figure 3.4.

As we can see, for GoldenEye, the average analysis time per malware is around 44 minutes, while the average time for Related Work I is 394 minutes, which is around 9 times slower. Furthermore, the worst case for GoldenEye never exceeds 175 minutes while there are 12% of tested malware takes longer than 12 hours for

Related Work I (note that if we do not set the 12 hour limit, the average for Related Work I will be much longer). This clearly indicates that GoldenEye is much more efficient.

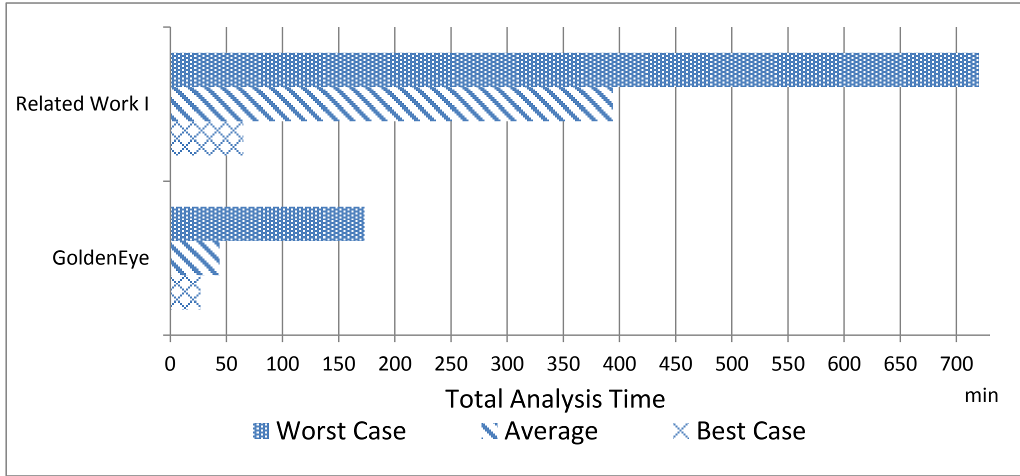

Figure 3.4: Analysis Time Comparison

In summary, it is evident that our approach has better performance regarding the trade-off of effectiveness and efficiency. We believe the main reason that other solutions have a higher overhead or lower effectiveness is because they are *not* designed to analyze malware's targeted environment. In other words, our approach is more *proactive* and *dynamic* to achieve the goal of targeted malware analysis.

3.3.4 Experiment on Known Targeted Malware Dataset

In this experiment, we aim to verify that our system can extract known targeted malware's targeted environment. We began our experiment from collecting the ground truth of some malware set. We looked up multiple online resources, such as [63], for the documentation about our collected malware samples. In particular, we first verified that all of them have been used for some targeted attacks, which

means they all need to check some environments and then expose their real malicious intention. Secondly, we manually examined their analysis report and summarize their interested environment elements. We grouped them into five categories: *System Information*, *Network Status*, *Hardware*, *Customized Objects*, and *Library/Process*. For instance, if one sample's malicious logic depends on some system-wide mutex, we considered it as sensitive to *Customized Objects*. We recorded our manual findings about our test dataset in Table 3.3(a).

	System	*Network*	*Hardware*	*Customized Object*	Library Process
Conficker[63]	√	√		√	√
Zeus[86]	√	√	√	√	√
Sality[85]	√	√			
Bifrost[55]	√	√	√	√	
iBank[81]	√	√	√	√	√
nuclearRAT[84]	√	√	√	√	√
Duqu[80]	√	√	√	√	√
Nitro[21]	√	√	√		
Qakbot[9]	√			√	√

(a) Ground Truth

	System	*Network*	*Hardware*	*Customized Object*	Library Process
Conficker	√	√		○	√
Zeus	√	√	√	√	○
Sality	√	√			
Bifrost	√	√	√	○	
iBank	√	√	×	×	√
nuclearRAT	√	×	√	○	○
Duqu	○	√	√	√	√
Nitro	○	√	○		
Qakbot	○			√	√

(b) GoldenEye Environment Extraction Result
√: Correctly Extracted, ○: Similar Element
×: Not Extracted

Table 3.3: Test on Targeted Malware

There are several caveats in the test. First, if the documentation does not clearly

mention the sample's MD5 or the sample with the specific MD5 cannot be found online, it may bring some inaccurate measurement for the result. One example is the Trojan iBank [81] case. We analyze some of its variants and they may not exhibit the same behaviors as the documented states. Second, we conclude the extraction result in three types: (a) *Correctly Extracted* means GoldenEye can extract the exact same environment element as document states. (b) *Similar Element* means GoldenEye finds some element that acts the similar functionality as mentioned in the document, but such element may have different name as the document described. We suspect it is probably because the element name is dynamically generated based on different information. For this type, we consider GoldenEye successfully extracts the environment information, because the correct element name could be derived through further manual examination or automatic symbolic execution [71]. (c) *Not Extracted* means GoldenEye fails to extract the environment element.

From the result, we can see that our GoldenEye can correctly detect most of the targeted environment elements (41 out of 44) within the 5-min analysis time limit. However, our system fails to extract 3 elements out of 44 cases. After we manually unpack the code and check the reason of the failures, we find there are two main reasons: (1) Some hardware query functions are not in our labeled API list (e.g., in the case of iBank). This could be solved if we improve our labeled API list. (2) Some element check only occurs after the malware successfully interacts with a remote (C&C) server (e.g., in the case of nuclearRAT). However, these servers may not be alive during our test thus we fail to observe such checks.

3.3.5 Case Studies

Next, we study some cases in our analysis. We list several environment targets which may trigger malware's activities.

Targeted Location. For Conficker A, GoldenEye successful captures the `GetKeyboardLayout` system call and automatically extracts malware's intention of not infecting the system with Ukrainian keyboard[63]. For some variants of Bifrost[55], GoldenEye finds they query the system language to check whether the running OS is Chinese system or not, which are their targeted victim machines.

For these cases, GoldenEye can intelligently change the query result of APIs, such as `GetKeyboardLayout`, to make malware believe they are running in their targeted machine/location.

User Credentials. We found several malware samples target at user credentials to conduct their malicious activities. For example, we found that Neloweg[83] will access registry at `Microsoft/Internet Account Manager/Accounts key`, which stores users' outlook credentials. Similar examples also include Koobface[50], which targets at user's facebook credentials. GoldenEye successfully captures these malicious intents by providing fake credentials/file/registry to malware and allowing the malware continue execution. While the malware's further execution may fail because GoldenEye may not provide the exact correct content of the credential, GoldenEye can still provide enough targeted environment information to malware analysts.

System Invariants. In our test, GoldenEye extracted one mutex from Sality [85] whose name is `uxJLpe1m`. In the report, we found that the existence of such mutex may disable Sality's execution. This turns out to be some common logic for a set of malware to prevent multiple infections. Similar logic has also been found in Zeus [86] and Conficker [63]. For these cases, even though the clean environment, which does not contain the mutex, is the ideal environment for analysis, we can still see that GoldenEye's extracted information is useful, potentially for malware prevention.

Displayed Windows and Installed Library. iBank [81] Trojan is one example that is sensitive to certain displayed windows and installed library. In particular, GoldenEye detects that IBank tries to find the window `"_AVP.Root"`, which belongs to Kasperky software. Meanwhile, it also detects that IBank accesses `avipc.dll` in the home path of Avira Anti-virus software. Our GoldenEye further detects if such library or window exists, the malware exhibits more behaviors by calling the function `AvIpcCall` in the library to kill the AV-tools. IBank samples tell us that if our analysis is performed in an environment without AV tools installed, we will miss these anti-AV behaviors. Hence, as a side effect, GoldenEye could be a good automatic tools for analysts to detect malware's anti-AV behaviors.

Others. Last but not least, we always assume exposing more malicious behaviors is better. However, detecting some path with less malicious behaviors may be also interesting. One example we find in our dataset is Qakbot [9]. The malware exhibits some behaviors related to some registry entry. This malware tries to write `_qbothome_qbotinj.exe` into a common start up registry key `CurrentVersion\Run`. The further logic for Qakbot needs to check the existence of such registry entry and if it fails, malware goes to sleep routine without directly exhibiting some malicious behaviors. This case is interesting for us because we find that by changing environment setting, we could even observe some *hidden dormant* functionality. Discovering such hidden dormant functionality may help defenders to make some schemes for slowing down the fast-spreading of certain malware.

3.3.6 Experiment on Distributed Deployment of GoldenEye

Finally, we evaluate the performance overhead of our distributed deployment of GoldenEye. In this experiment, we measure three cases:

- Case I: Generate a parallel task for all environment-sensitive branches.

- Case II: Generate a parallel task only when the branch evaluation cannot decide a branch after measuring the branch selection heuristics.
- Case III: Do not generate a parallel task and do not conduct rolling back, i.e., using a single machine instead of distributed deployment (for undetermined paths, we select the default environment as desired).

We use additional four worker (virtual) machines for this measurement (Case I and II). Each virtual machine installs original unpatched Windows XP SP1 operating system. We randomly select 100 malware samples and run each sample for at most 300 seconds in each configuration. We compare performance with the baseline case, which is running each malware in the default environment.

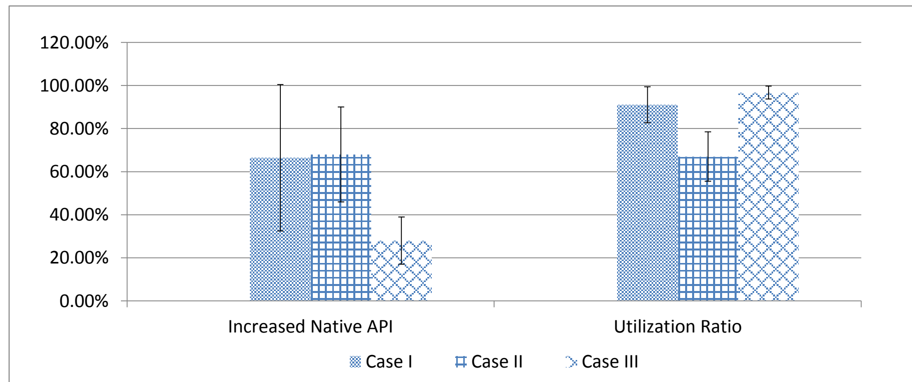

Figure 3.5: Measurement of Distributed GoldenEye

The result is summarized in Figure 3.5. As seen in the figure, we study the effectiveness by measuring the increased ratio of native APIs. As expected, Case I and II expose over 30% more behaviors than Case III. However, the standard deviation of Case I is higher than Case II. It shows that, with the same analysis time, the first approach may not outperform the second case because exploring all

environment-sensitive paths is not efficient enough. We also measure the utilization ratio of the analysis machine(s), which is defined as the percentage of time for an analysis machine to run the analysis task within the given 300 seconds. The average utilization ratio from VMs in Case I is over 90%, which is much higher than Case II. In short, we conclude that Case II configuration of GoldenEye, i.e., combining the branch selection scheme with the distributed deployment, seems to achieve the best balance between effectiveness and resource consumption among the three cases.

3.4 Limitation

As a new step toward systematic targeted malware analysis, our solution is not perfect and we discuss limitations/evasion below.

3.4.1 Correctness of Path Selection/Prediction

One limitation of our approach is that the correctness of our branch evaluation depends on whether malware's behavior fits our heuristics. One solution for this problem is to explore all possible branches by multi-round snapshot-and-recover analysis, as in [57]. However, this scheme may cause much higher overhead because of the *path explosion* problem. Hence, to trade off the performance, we choose to apply snapshot-and-recover only when we cannot apply the heuristics.

3.4.2 Possible Problems of Taint Analysis

In our scheme, we apply taint analysis at the stages of preprocessing and speculative execution. For preprocessing, taint analysis can help us filter out the malware which are not sensitive to the environment. For speculative execution, taint analysis helps to save execution overhead from multiple aspects. However, as discussed in related work [71], taint analysis could have limitations of *over-tainting* and *under-tainting*. Even though it may cause the problem of imprecise results, for our cases,

the limitation can seldom affect our analysis. This is because: (1) Even over-tainting costs more overhead for speculative execution, our scheme is still more lightweight than existing approaches. (2) The under-tainting problem may mislead our branch prediction. However, by using stricter branch selection criteria, we could avoid such wrong branch. Meanwhile, conducting more roll-backing operations on some critical branches can also improve the overall accuracy. (3) Our analysis can be independently conducted even without taint analysis. In this case, our speculative execution engine has to be executed at all branches to truncate undesired environments. Even though it may cause more overhead, we believe it still outperforms other approaches because it prevents unnecessary rolling-back.

3.4.3 Evasion through Misleading the Analysis

By knowing our heuristics for branch selection, the attacker could mislead our analysis through injecting some certain APIs in the branches. However, some heuristics (e.g., environment interaction, process termination) are relatively hard to be evaded because otherwise they will be against the malware's execution intention. We note that even in the worst case (we have to rewind to explore another branch, similar to existing multi-path solutions), our solution is still better than a blind multi-path exploration scheme.

Another way to evade the analysis is to query environment information and process it at a very later time. To handle this issue, we could increase the capacity of parallel spaces and track the tainted environment elements throughout the whole analysis by paying a little more analysis overhead.

Malware can insert some dormant functions such as `sleep` because GoldenEye may not prefer to choose branches in which malware could enter a dormant status. To handle such cases, GoldenEye can examine more code blocks in the foreseeing

operation in order to make a more accurate branch selection or could simply generate a parallel task for another worker machine.

Last but not least, current implementation of GoldenEye does not handle implicit control flow, a common issue to many dynamic analysis systems. Hence, malware authors may evade the analysis by including implicit control flow. However, this issue could be partially solved by conducting symbolic execution on indirect branches. We leave it as our future work.

3.5 Related Works

Among many studies on binary analysis using forced execution techniques [48, 28, 72, 5, 18, 53, 98, 93], one closely related work to GoldenEye is the approach that explores the execution paths of malware binaries. The paper [57] demonstrated a multi-path exploration scheme by combining enforced execution and snapshot recover. Brumley *et al.* [11] proposed an approach that applies taint analysis and symbolic execution to derive the trigger condition of malware's hidden behavior.

Lindorfer *et al.* [53] discussed an approach to detect and analyze environment-sensitive malware by running samples in *multiple statically* configured environments. In contrast, our approach is more intelligent as it *dynamically* constructs an adaptive environment for each targeted malware. Meanwhile, they only consider the system configuration as the environment, but we extend the environment definition to include system objects.

3.6 Summary

In this chapter, we present a dynamic analysis system, GoldenEye, to analyze new malware. From the discussion of new malware's characteristic, we propose to develop an automatic approach to extract malware targeted environment before the traditional dynamic analysis. To serve this goal, we design and present sev-

eral dynamic analysis techniques, such as parallel environment spaces construction, speculative execution in parallel spaces and branch evaluation to solve the technique challenges of the problem. In the evaluation, we further show our schemes can work on real-world malware corpus and achieve a better performance tradeoff compared with the existing works. Last but not least, our initial work may stimulates more following research to discuss more systematic method for analyzing targeted malware threat.

As another important discovery in the evaluation result, we find malware commonly applies Infection Marker as a mechanism to prevent duplicate infection. In the next chapter, we will have a detailed discussion about this finding.

4. ANALYZING AND EXTRACTING MALWARE VACCINES FOR HOST PROTECTION

4.1 Introduction

Malware is a severe threat to our computer systems. To combat malware, the state-of-the-art defense at end-hosts mainly focuses on detection techniques, which often fall into two categories: signature-based detection and behavior-based detection. A signature-based approach typically attempts to extract some unique string patterns from malware binaries. Unfortunately, the signature generation and update speed usually cannot keep up with the quickly increasing malware samples each day in the wild due to the wide use of polymorphisms/packers in malware. While a behavior-based approach could be relatively more stable in terms of detecting the same set of malware and their variants, it is typically very expensive and may cause a noticeable performance overhead on end hosts.

Therefore, the need of new lightweight and complementary techniques for effective malware defense is still pressing. Interestingly, we find malware infection works similarly to pandemic diseases. Since a widely used approach to prevent further infection of our human beings from the same disease is through injecting vaccines, if we were able to generate vaccines for a piece of malware, we would have been able to prevent it from infecting a wider range of machines (considering the case of botnets). Fortunately, we find malware often contains system-resource-sensitive condition checks or constraints to avoid any duplicate infection, make sure to obtain required resources, or try to infect only targeted computers, etc. For instance, many fast-spreading malware programs (e.g., Conficker [63]) will clearly mark an infected machine as *infected* such that they can avoid wasting time and effort in re-infecting

the machine. As such, this *infection marker* can be considered as an effective and safe vaccine to immunize a clean machine from the same infection.

In general, any system resource/environment variables that are directly or indirectly used in path conditions (such as registry, mutex), or those that lead to the failure of certain system calls, can all be considered for vaccine generation, because these external environment state can impact the behavior of the malware. While it might lead to an over approximation by considering all these state variables, we can run vaccine tests to eliminate the mistakenly classified environment variables, similar to the biological vaccine test in the real world.

Based on the above observation, we propose AutoVac [99], a new technique to automatically generate vaccines for effective and efficient malware immunization from the same infection. While theoretically manipulating any variables that lead to a conditional check of malware execution could potentially be used as a vaccine, we would like to focus on the variables whose states can be controlled by the external environment such as registry, certain file names, etc. As such, the environment resources accessed by malware are of our interest. Specifically, we design a program analysis technique to determine whether the manipulation of these resources can successfully prevent malware's infection/execution. We treat such resources as our malware vaccines and derive concrete information needed for generating vaccines. After we generate the vaccines, we then inject them into end hosts.

4.2 Problem Statement

The concept of vaccine is originated from biology. It refers to a biological preparation that improves immunity to a particular disease by injecting certain agent that resembles a disease-causing microorganism. The malware vaccine idea was initially mentioned by David Ferbrache in his 1992 book [34]. As stated in the book, *"With*

the computer environment fragments of viral material may also be used-in this case the signature recognition strings which the virus uses to prevent repeated replication. These fragments may safely be added to existing cells and will protect against the virus." Unfortunately, he only briefly talked about this high level features of vaccination and did not systematically explore this problem further.

Throughout 20 years evolution of malware, when we revisit the vaccination idea, we realize that we can further explore this problem in the new context of complex malware (e.g., targeted malware) defense. From our viewpoint, a malware vaccine is a computational preparation that improves immunity to a particular malware program. Essentially, malware, like any generic program, usually conducts a series of operations on system resources and outputs the computation result. These system resources in a computer system are analogue to the microorganisms in our body.

Thus, we define a malware vaccine as a specific system resource (or a collection of them) that is created or used by malware in order for its normal infection and execution. Such malware vaccine typically has two kinds of behavior:

- It simulates the existence of certain computer organism (system environment/resource) such that malware will exit upon the awareness of such existence (because it does not want to re-infect the victim again, or the victim does not have a targeted environment, etc.).
- It prevents malware from creating/accessing certain critical computer organism such that malware cannot obtain its essential resources to fulfill the functions.

Besides the aforementioned mentioned categories of malware vaccines, we can further define different vaccine types from different perspectives.

First, from the perspective of identification, the vaccine *identifier* is defined as a combination of *resource type* and *name* of malware-targeted resources. To avoid

vaccines' unwanted side effect to benign software running on end-host, the vaccine identifier should be as *unique* and *deterministic* as possible. Thus, in our taxonomy, an identifier can be categorized as: static (e.g., constant value), partial static (e.g., it conforms to a specific regular expression), or algorithm-deterministic (e.g., it is calculated with customized algorithms).

Similar to biological vaccines that may not guarantee the complete protection from a disease, the effectiveness of a malware vaccine can vary. Based on the effectiveness, we can classify malware vaccines into two types: full immunization that can completely cease the malware execution (e.g., negating the first few condition checks to prevent any malicious behavior execution), and partial immunization that significantly affects the execution of some major functions in malware (e.g., malware is not able to keep persistent in the system if rebooted, or malware is not able to perform key network communication such as C&C, self-updating).

In terms of vaccine delivery and deployment, there could be two categories: direct injection and creation of vaccine daemon. Direct injection is very lightweight, e.g., a specific `mutex` name or file name, and the vaccine can be simply injected into the target computer once and it will be effective afterwards. Vaccine daemon requires running a service program (i.e., a daemon) on the targeted machine, and such daemon can prevent the creation (or other access types) of certain specific files, registries, libraries, system services, windows, processes to further prevent malware from obtaining critical resources or information to fulfill its functionalities (such as for partial immunization).

It is worth noting that an ideal malware vaccine is those with full immunization and one-time direct injection. However, other types of vaccines are also useful, as discussed later and shown in our evaluation (§4.4).

As a complementary technique to existing malware defense, vaccines may not be

used to protect machines from all malware attacks. However, they can be used for current, high-profile, large-scale malware propagation and infections, which may last for a period of time, e.g., several days, weeks, or months. If we can capture the binary at the initial infection stage, we can quickly generate vaccines and protect our uninfected machines from the attacks, until a better detection or prevention solutions (e.g., a system/software patch to fix the vulnerability) are available and fully deployed.

4.3 System Design

The overall design of AutoVac is shown in Figure 4.1. In the following section, we discuss the details about our design.

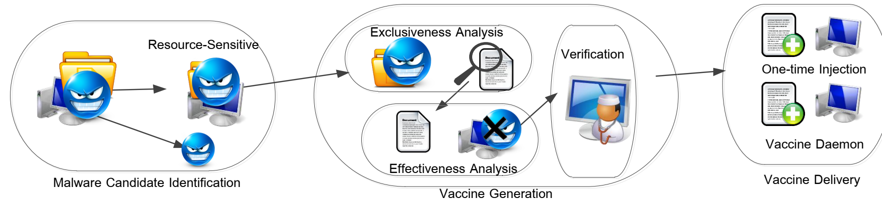

Figure 4.1: System Architecture of AutoVac

4.3.1 Phase I: Candidate Selection

Given a malware sample, AutoVac first determines whether it is possible to generate a vaccine, and at the same time collect the behavior information to facilitate the next step analysis. Since our vaccine is essentially composed of system resources that have a direct or indirect (through propagation) impact on the malware execution, we adopt a variant of dynamic taint analysis [71] to achieve this.

Taint Sources. Taint sources define the origins of tainted data. Our current focus is on those system-resource-related data that can possibly impact the malware behavior. However, there is a wide range of system resources and certainly some of them cannot be used such as system-assigned random objects. As such, we have to systematically study these resources and identify our taint source. In particular, we use the following criteria to decide whether a system resource should be tainted.

- **Unique Presence.** Our focused system resources should be commonly used by malware, and these resources should be *uniquely identified*. Thus, those *transient* system resources, e.g., events, signals, critical sections, are out of our interest.
- **Less Impact to Benign Software.** Our targeted resources should have *little or minor impact* to benign programs. This requirement would exclude many system-wide objects and information, such as timers, performance counters, input/output devices, removable devices, because they are commonly accessed by benign programs
- **Easier Deployment.** Our targeted resources should be lightly deployed onto end-hosts as vaccines. To this end, injecting some specific files or `mutex` into the end-host would be viable options. Therefore, files, `mutex`, or registry will be our main targeted resources.

API Labeling. After applying the above criteria, eventually `mutex`, static files, and registry items are of our particular interest. Meanwhile, the propagation use of these resources such as process, library, GUI window and services are also of our interest because these resources depend on some deterministic resource identifiers. However, at the instruction level, these *resource-identifiers* often get accessed through

system APIs. Thus, we have to examine each Windows API to define our taint sources.

More specifically, all the system resource access APIs (e.g., `NtQueryObject`) are of our interest. AutoVac taint the return values as well as the affected arguments of these functions. In our design, we examined over 800 windows APIs and we classify them into the following two categories.

- *Tainting the return value* Most APIs only affect the return values (always stored in `EAX`), such as `OpenMutex`, `NtSaveKey`. For them, we just taint the return value.
- *Tainting the argument* Some APIs store the affected values in the arguments. For instance, `NtOpenKey` and `NtOpenFile` store the return handler in their first parameters.

Besides tainting the return values or arguments, we also need to record the concrete values of the arguments to these APIs because eventually our vaccines work by affecting the system environments which are their arguments. Meanwhile, not all the arguments are of our interest, and only those *resource-identifiers*. This is also a tedious procedure to identify these *resource-identifiers*. Table 4.1 shows an example on how we label the two Windows APIs.

4.3.2 Phase II: Vaccine Generation

Once a malware sample has been flagged to "possibly have a vaccine" in Phase-I, it is fed to our Phase-II to perform a deeper analysis, including exclusiveness analysis, impact analysis, and determinism analysis.

Exclusiveness Analysis. The goal of our exclusiveness analysis is to exclude the resources that have been used in benign software. For instance, some resources such

	`OpenMutex`	`ReadFile`
Resource Type	Mutex	File
resource-identifier	3rd parameter: *lpName*	1st parameter: *hFile for Handle Map*
Success	`EAX`: Valid Handle Value	`EAX`: TRUE
Failure	`EAX`: NULL, `GetLastError: 0x02`	`EAX`: FALSE `GetLastError: 0x1E`

Table 4.1: Labeling Examples for OpenMutex/ReadFile

as library names `uxtheme.dll, mscrt.dll` could be used in benign programs. We must exclude them otherwise our vaccine will have false positives.

In Phase-I, AutoVac has logged all the *resource-identifiers*, and next we would like to query whether or not each *identifier* is unique to the malware.

Our basic idea is inspired by a Googling approach used in previous studies [79]. Essentially we use Google query APIs to search *resource-identifiers*. Based on the return results and their context, we infer whether these resources are already associated with benign software. We refer readers to [79] for more details.

In short, from our search query, if the *resource-identifiers* does not conflict with benign software or there is no any matching search result, then we proceed with further analysis.

Impact Analysis. Given a list of the system resources that can (in)directly affect the malware execution and the corresponding APIs provided in Phase-I, AutoVac will run the malware again in a controlled environment such that we can mutate the return value or involved arguments, and test whether malware will exhibit different behavior or not. Our current design is to mutate each involved API one at a time, and compare the behavior with our normal execution captured in Phase-I.

Trace Differential Analysis. Then the next question is how we compare the malware behavior in two traces: one is a normal execution, and the other is a resource mutated execution.

Finding the differences in two traces has been discussed in previous literature (e.g., [102, 46]). It is essentially a *program alignment* problem [102]. The basic idea is to align two execution points that are equivalent to each other and then compute the differences only between the *unaligned* instructions. In our scenario, we try to obtain the high-level information such as whether the malware will terminate rather than the minor instruction level execution differences. Thus, in our design, we use the API call sequences (as we have already logged all the executed APIs and their calling context information), and present an API sequence alignment algorithm as shown in Algorithms 1.

In particular, we adopted an alignment algorithm from Zeller [102], which uses the *execution context* for each instruction for the comparison. If the instruction and its execution context are equivalent (line 4), they are aligned together. However, we do not need to compare instruction by instruction, but rather at the granularity of APIs. Thus, we define a calling execution context as a *triple*:

<*API-name*, *Caller-PC*, *Parameter list*>

For the *parameter list*, we only compare the *static* parameters that are *identical* across different executions. Note that all these information has been logged either in Phase-I for the normal execution, or logged in Phase-II for the mutated execution. Also, the reason we have to log the *Caller-PC* is for the preciseness.

As illustrated in Algorithm 1, our analysis begins from the start of the trace, then proceeds with a linear searching for each system/library call in the mutated trace, and examines whether it could be aligned with some call in the normal run trace

Algorithm 1: Differential Analysis on the API-Call Traces

$\prod_m$: Manipulated Call Trace, $\prod_n$: Natural Call Trace
Δ_m: Unaligned Call Trace in $\prod_m$, Δ_n: Unaligned Call Trace in $\prod_n$,
$f_{\prod}$: $\langle name, caller\ eip, parameter\ list \rangle$, f_Δ: $\langle name, parameter\ list \rangle$

1: $\Delta_m \leftarrow \emptyset$,$\Delta_n \leftarrow \emptyset$
2: **for** call $f_{\prod_m}$ in $\prod_m$ **do**
3:     **for** call $f_{\prod_n}$ in $\prod_m$ **do**
4:         **if** isAligned($f_{\prod_m}$,$f_{\prod_n}$) **then**
5:             GOTO FIND_ALIGNED
6:         **end if**
7:     **end for**
8:     $\Delta_m = \Delta_m \bigcup f_{\Delta_m}$
9: **end for**
10: $\Delta_n = \prod_n$
11: FIND_ALIGNED:
12: $\Delta_n = \prod_m [0, index(f_{\prod_n})]$
13: $\{f_{\Delta_i}\}$=Diff(Δ_m,Δ_n)
14: return $\{f_{\Delta_i}\}$

(line $2-8$). If we find an anchor point, we generate two difference sets Δ_m and Δ_n.

Next, we examine the two Δ sets to evaluate the further differences, and classify the vaccine immunization type. Specifically, we define three kinds of immunization effects.

Full Immunization. If we find APIs such as `ExitThread`, `TerminateProcess`, and `TerminateThread` in Δ, then certainly the mutated system resources can be served as a full immunization vaccine, because the malware has killed itself.

Partial Immunization. Some vaccines may significantly weaken certain important functions of malware. We consider them as partial immunization vaccines. More specifically, we currently focus on the follow four types of partial immunization:

- *Type I: Disable Kernel Injection* An important malicious function of malware is to raise its privilege. The common way they use is to inject a kernel driver into an end host. There are several system calls (mainly undocumented) such as `OpenSCManager` have been used for this. Furthermore, some malware com-

monly copies itself as a new file with its name ending with `.sys`, which implies that some kernel driver is created by the malware.

- *Type II: Disable Massive Network Behavior* If we find the normal execution is full of network-related functions, while the manipulated execution is clean from such calls, we consider such vaccine as **Type-II** Partial Immunization.

- *Type III: Disable Malware Persistence* Malware typically modifies specific registry entries such as `Run` subkeys in mutliple register paths. Other autostart approaches include (a) file operations on `startup` folder or `system.ini` files, (b) creation of new service entries, (c) access of `winlogon` binary. Through differential analysis we can tell if these operations are lost in the mutated execution while present in the normal execution.

- *Type IV: Disable Benign Process Injection* To be more evasive, malware often inject themselves into some benign processes. Processes such as `explorer.exe` and `svchost.exe` are common targets. If we find such a clear pattern in the differential analysis, we consider these vaccines as Type-IV Partial Immunization.

No Immunization. If none of the above APIs are in the Δ, then we classify this vaccine with no effect to stop or affect malware behavior.

Determinism Analysis. We next need to verify the determinism of the extracted resource-identifiers.

Backward Taint Tracking and Program Slicing. Given a resource-identifier, we need to identify whether it is deterministic or entirely random. We choose to trace the root-cause for the generation of the resource-identifier.

To back track the procedure of how malware generates an identifier, we perform a backward taint tracking. The basic idea is to include all the instructions that have contributed to the creation of the resource-identifier, which is the argument of the API of our interest. To this end, starting from data-use of the argument, we back track each executed instruction to check whether or not their operands have been involved to define the data. If so, we taint the source operand as the same symbol and continue the backward propagation. We perform the analysis offline on logged traces.

The termination of our backward tracking is the point to identify the root-cause that generates the identifier's name. We continue backward propagation until tainted source is either from read-only regions (e.g., static strings), or constant values, or the return value of the system APIs. Based on these different sources, we decide whether the generation of the identifier is deterministic or not.

An identifier has a *non-deterministic* type if and only if *all* elements of its composition are resulted from some random functions (e.g., `GetPerformanceCounter` and `GetTempFileName`).

As illustrated in the left part of Fig 4.2, if the termination data point is from a read-only segment such as `.rdata`, or constant values, we can easily mark it as *static*. Similarly, if an identifier is constructed using some non-deterministic value combined with some constant value, we can mark it as *partial static*, and such an identifier will be deployed using a slightly different strategy compared to the scenario of purely static identifier.

An identifier could be *algorithm-deterministic*, namely, its identifier is generated through certain computation. Some appear-to-be random name can be generated from some invariable seed, such as computer name or hardware serial number. Algorithm-deterministic names will be backward propagated to some semantic-

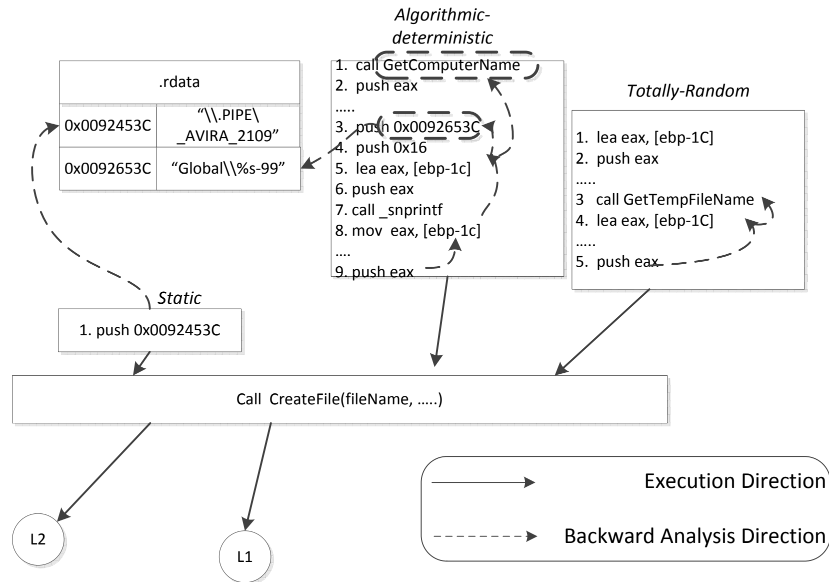

Figure 4.2: Sample Malware Code and the Traced Behavior

known APIs. We use these APIs to decide the root-cause type when generating the name. One example is shown in the middle part of Figure4.2. We use the `GetComputerName` to infer that the input should be a computer name.

For such algorithm-deterministic identifier, we also need to find the generation logic because we need to replay and compute it for each end-host. We apply the existing backward program slicing[49] techniques to extract an independent, executable program slice for that.

At the end of this step, we delete all the entirely random (non-deterministic) identifiers.

4.3.3 Phase III: Vaccine Delivery and Deployment

After we generate the vaccine, we next describe how to deliver and deploy the vaccines to an end-user computer.

Direct Injection. Direct injection works for static identifiers. If a vaccine stops malware execution by frustrating the presence checking of static type of resources, we inject it by creating or deleting the resources. For instance, if the malware needs to open certain static file (or registry) before proceeding the malicious functionality, then we remove the static file (or registry), or vice versa. Moreover, we accordingly adjust the injected file's access privilege to disallow certain operation such as read and write. In these cases, when a low-privilege malware program attempts to access a resource, which is a common case at the initial infection stage, static vaccines efficiently stop further malicious behavior.

Vaccine Daemon. Vaccine daemon works for algorithm-deterministic identifier and partially static identifier. For an algorithm-deterministic identifier, we have extracted a program slice of the resource-identifier generation logic with knowledge about its input, such as a computer name or an IP address. To generate the vaccine, we collect these information ahead and run the captured program slice. Such procedure works very similar to Inspector Gadget [49]. Our daemon process runs periodically to check whether the input has been changed and the vaccine needs to be re-generated.

Vaccine daemon is also designed for identifying resource name represented using regular expressions (i.e., distinguishable partial static vaccines). Specifically, at the end host, we dynamically intercept the APIs and resolve their resource-identifiers. If the daemon monitors that a resource identifier matches with our partial static vaccine, it will return the predefined result to stop the malware execution.

4.4 Evaluation

We have implemented AutoVac. While our online dynamic analysis can be implemented using virtual machine monitors such as TEMU [100], we use Dy-

Category	# Malware	Percentage
Trojan	184	10.72%
Backdoor	722	42.07%
Downloader	574	33.44%
Adware	73	4.25%
Worm	104	6.06%
Virus	59	3.43%
Total	1,716	100%

Table 4.2: Malware's Classification from VirusTotal

namoRIO [31] to implement due to its simplicity and flexibility in binary instrumentation. Our differential analysis module is implemented using offline parsing of the execution logs. Also, to perform taint analysis we translate the X86 instructions into an intermediate language *BIL*[12], and then we develop our own parser code to identify the resource-sensitive branches and perform differential analysis. Our exclusiveness analysis involves a search engine query component, for which we implement using the the API provided by Google. In this section, we present our evaluation results.

4.4.1 Experiment Dataset

Our test dataset consists of $1,716$ malware samples, which are collected from multiple online malware repositories (e.g., [3, 61, 82]) with mostly from Anubis [3]. We also leverage an online malware classification tool, *VirusTotal* [88], to obtain the classification information for these malware. We summarize classification results in Table 4.2. We can see that these malware samples fall into 6 categories such as Backdoor (722 samples), Downloader (574 samples) and Trojan (184 samples).

All the experiments are running in machines with Intel Core Duo 1.50GHz CPU and 8 GB memory.

4.4.2 Experiment on Vaccine Candidate Selection

In the first step, we monitor malware's access to system resources. We conduct this experiment by running these $1,716$ malware samples in our analysis environment and each sample runs for 1 minute (we tend to believe the resource checks usually happen in the early stage of the malware execution and we thus choose this 1 minute threshold). We hook 89 system/library calls as tainted sources that are related to resource operations. The resources in our evaluation include *file*, *mutex*, *registry*, *window*, *process*, *library* and *service*. We measure the basic operations for these resources such as read/write for file and registry, open/create for other resources. Meanwhile, for each execution instance of the hooked function, we examine their callers' `PC` and make sure it does not belong to the system library's address space. Thus, we do not count the functions that are called inside the system/library calls.

For $1,716$ malware samples, we successfully tracked $460,323$ occurrences of these API calls. Through our taint analysis in this phase, we identified that $371,015$(80.3%) occurrences of the calls will possibly deviate the execution of the malware samples. This result confirms that real-world malware is indeed resource sensitive.

Among these $371,015$ occurrences, we further made a statistic study based on the resource type and its corresponding operations. The result is shown in Figure 4.3. From the figure, we can see that around 37.39% of the resource accesses account for file operation. Mutex (7.07%) and registry (20.08%) are also commonly accessed by malware. We consider these three types of resources can be efficiently delivered using the injection scheme. Meanwhile, malware's logic is also commonly sensitive to other types of resources such as windows (13.14%), process (8.02%), library (6.6%) and service (3.4%).

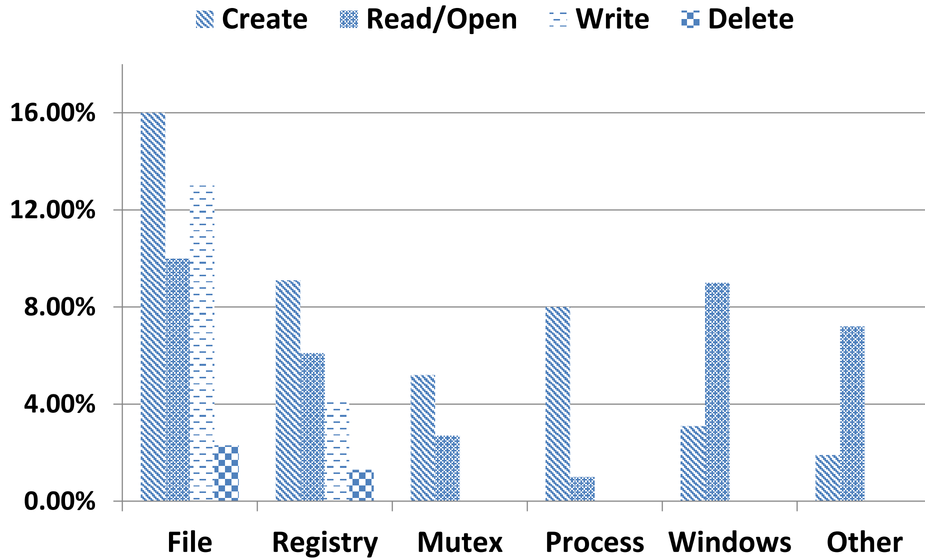

Figure 4.3: Statistics on Malware's Resource Sensitive Behaviors

4.4.3 Experiment on Vaccine Generation

In the evaluation, we analyzed all 1,716 malware in a controlled environment.

In total, we generated 536 vaccines that belong to 210 malware samples. The result is presented in Table 4.4. For each column, we classify the vaccines as full immunization or partial immunization (Type-I to Type-IV). We also list the statistics on the vaccine distribution among different resource types in Table 4.4. Among all vaccines, we find 373 vaccines have static identifiers, and 163 samples have *algorithm-deterministic* or *partial static* identifiers.

To zoom-in the details of these vaccines, we select 10 representative samples and describe them in Table 4.3. We can see that most of these vaccines stop several logic of malware's infections. In some cases, different operations on the resources can even cause different effects on malware's logic. For example, for the last malware in Table 4.3, we find that the failure of *creating* a file will stop malware's process

Seq	Type	OperType	Impact	Identifier	Malicious Sample Md5
1	Mutex	E	T	!VoqA.I4	df1df624c5da833d3882d22a2e2456c9
2	File	C,R,W	P,H	%system32% \twinrsdi.exe	1b6fb589f36654af0ef44aa92f94324a
3	File	C,E,R,	P,H,N	%system32% \dwdsregt.exe	24784256bbbb936dc1e0999c307883c8
4	File	C,E,R,W	K,P	%system32%\driver\qatpcks.sys	27d18e20e253391112d50b2b49440aea
5	Mutex	E	T	GTSKISNAUOI	ee5878eab962b032c78c1d6eec7ec917
6	Mutex	E	P,H	fx221	af48ecfcc1812d6f814a26792107b80e
7	Mutex	C,E	T	)ryt-24qtqq26sn]9c	b534b75da5fc3b9b178c60bf10b1feca
8	Mutex	C,E,R	P,H	_AVIRA_2109	04a93b1f08a1675c67c9975a7024c3d6
9	File	C,E,R,W	P,H	%system32% \ shlmon.exe	af48ecfcc1812d6f814a26792107b80e
10	File	C,E,R,W	T,P	%system32%\sdra64.exe	04a93b1f08a1675c67c9975a7024c3d6

Table 4.3: Vaccine Samples (Operation Type Symbols - check Existence (E), Create (C), Read (R) and Write (W), Impact Symbol - Termination (T), Process Hijacking (H), Persistence (P), Kernel Injection (K) and Network Massive Attack (N))

Resource	Full	Type-I	Type-II	Type-III	Type-IV	All
File	31	19	17	110	61	238
Registry	10	11	3	72	19	115
Mutex	5	3	3	16	3	30
Process	2	5	2	18	5	32
Windows	0	4	3	8	3	18
Library	19	5	1	10	19	54
Service	7	4	0	17	21	49
Total	74	51	29	251	131	536

Table 4.4: Evaluation on Vaccine Generation

hijacking logic, and the failure of *writing* a file will crash the malware process.

For the generated 536 vaccines, we also combined their types with the 210 malware's classification information to see what is the common vaccine type for different kinds of malware. The result is shown in Table 4.5. From this table, we can see that the file resources are the common vaccines for many malware families. Meanwhile, the windows resource vaccine is better suitable for adware because the windows resource vaccine is attempting to prevent adware from creating their malicious windows. If such operations fail, adware will possibly stop their further action. Last but not least, mutex vaccine works better for worm and backdoor malware. This is also reasonable, because these malware highly depends on the mutex to prevent duplicate infection.

Vaccine Type	Backdoor	Trojan	Worm	Adware	Downloader	Virus
File	33%	27%	24%	30%	45%	81%
Registry	15%	29%	21%	13%	20%	19%
Windows	3%	14%	0%	47%	11%	0%
Mutex	8%	12%	29%	0%	2%	0%
Process	8%	7%	14%	0%	10%	0%
Library	26%	9%	4%	0%	7%	0%
Service	7%	2%	8%	10%	5%	0%
Deployment						
Direct	67%	79%	63%	69%	69%	84%
Daemon	33%	21%	37%	31%	31%	16%

Table 4.5: Vaccine Statistics on Different Malware Families

We also report the statistics of our vaccine delivery for these 536 vaccines. As shown in Table 4.5, direct injection is the most common way to deploy vaccines on end hosts. Also, only about 20%-30% vaccines need a daemon for the deployment.

4.4.4 Case Studies

Next, we present two representative case studies to illustrate in greater details on how each of our resource access based vaccines can be used for malware infection immunization.

File-based Vaccines. One vaccine for Zeus/Zbot [86] family is a static file named `sdra64.exe` which is stored in the `system32` directory. We observe that if Zeus successfully creates this file, it will continue writing malicious bytes into that file using bytes in its resource and start a new process using this file.

Delivery: We deliver a vaccine by deliberately creating `sdra64.exe` at an end host. This file is owned by a super user and does not allow any creation operation by others. In this way, our vaccine prevents Zeus's attempt to start the malicious process.

Mutex-based Vaccines. One mutex vaccine is for Conficker, which is an algorithm-deterministic vaccine. This mutex vaccine can efficiently stop Conficker's infection

at its initialization stage.

Several other mutex examples include `_AVIRA_21099`, `_AVIRA_2109`, `_AVIRA_2108`, which belong to Zeus/Zbot[86] malware. This set of vaccines can stop multiple malware logic such as kernel injection, process hijacking, and network communication.

Delivery: Direct injection is an efficient approach to deliver mutex vaccines. We simply create a deterministic `_AVIRA_` mutex in the system to prevent Zbot's injection. For Conficker, we run the vaccine slice once at the end host and generate the mutex name for each computer.

4.4.5 Experiment on Vaccine Effectiveness

In this test, we evaluate the effect of our vaccines on the malware samples. As reported in §4.4.3, our vaccines can stop or weaken 210 samples' malicious behaviors. In this test, we run these 210 samples in both vaccine-deployed environment and the normal infection environment for 5 minutes. Then we compare the differences of their native system calls (all the NT native calls) in these two environments. We define a metric Behavior Decreasing Ratio, $BDR = \frac{N_n - N_d}{N_n}$, where N_n is the number native system calls in the normal environment while N_d is that number in the vaccine-deployed environment. The larger BDR is, the more reduction of functions by the vaccines. In Figure 4.4, we report the distribution of BDR according to different vaccines' effectiveness type.

From this figure, we can see that the full immunization vaccines are obvious the most effective ones and they all terminate the execution of malware (the reason why their BDR is not 100% is simply because of their initial executions before exit also have some native system calls). Our partial immunization vaccines all effectively achieve their goals by disabling key functions in the malware (through a careful manual examination, we confirm that all unwanted malicious logic has been disabled).

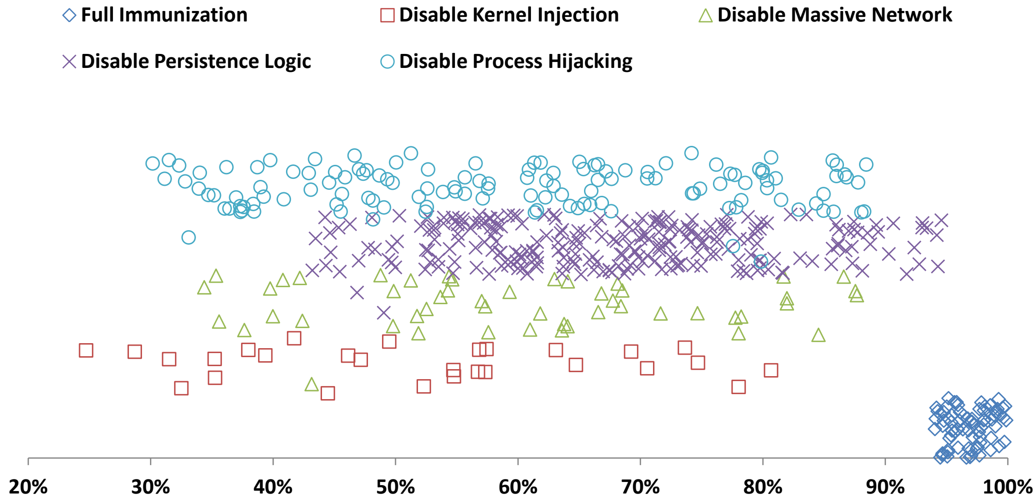

Figure 4.4: Distribution of BDR

One such example for Zeus is shown in Table 4.6. Even in the *worst* case in terms of BDR, our partial immunization vaccine can still reduce at least 24% malware's *important* system call activities. Note that BDR will certainly increase if we keep running the malware sample in a longer time period.

To further verify that our vaccines are effective for different variants in the same malware family, we choose 6 high-profile malware samples and perform another test. These samples are high-profile malware such as Conficker, Zeus/Zbot, and Sality, and for these 6 samples we have extracted a total of 17 different vaccines in our previous test. We then further collect 5 variants (binaries are different from what we have collected in the original dataset) belonging to each family (thus 30 new variants in total). Then we run the 30 new collected variants in both normal and vaccine-injected environments, similar to the previous experiment. We carefully analyze the execution differences and manually verify whether the injected vaccines have achieved the goal or not. The result is showed in Table **??**. Note that the 4th column indicates

Malware	Vaccine	Type	Impact Description
Zeus/Zbot	_AVIRA_2109	mutex	Stop process hijacking

Table 4.6: Example of a High-profile Malware Vaccine

Malware	Vaccine#	Type	Ideal Case	Verified	Ratio
Zeus/Zbot	6	mutex, file	30	23	77%
Conficker	2	mutex	10	10	100%
Qakbot	2	registry	10	10	100%
IBank	1	file	5	5	100%
Sality	3	mutex,file	15	12	80%
PosionIvy	3	mutex,file	15	10	67%
Total	17		85	70	82%

Table 4.7: Vaccine Effectiveness Evaluation on Malware Variants

the number of malicious functions that can be stopped if ideally these vaccines work for all variants, the 5th column indicates the actual number from our test, and the 6th column shows the percentage of success.

From the result, we can see that overall our vaccines can take effect in almost all variants. However, we do find that some vaccines can work for some variants but fail on others. One example is the file vaccine `sdra64.exe` which we did not find its use in 2 other Zbot variants. Fortunately, for each malware, we have extracted more than one vaccines. Thus, even some may not be effective for all variants, the combination of these vaccines can still achieve satisfiable results. We believe this test also highlights the importance of using an automatic tool (such as our AutoVac) to analyze malware samples to extract as many vaccines as possible, a goal otherwise very hard to achieve through manual analysis.

False Positive Test. Our next test is on the false positive evaluation, i.e., whether our generated vaccines will affect the normal program executions.

First, we install 5 different virtual machines running over 40 benign software (which includes the most common software typically seen on normal users' computers

such as all kinds of browsers, programming environments, multimedia applications, Office toolkits, IM and social networking tools, anti-virus tools, and P2P programs).

Then we equally inject our vaccines into each test machine and monitor their system logs over a period of a week. The result shows that our vaccines did not cause any problem to our running environments.

One could argue that this automatic test may underestimate users' interaction. Hence, we conduct another test to install 200 vaccines on 4 lab machines. All these four machines are for normal everyday use. The result also shows that our generated vaccines did not cause any trouble for the operation of existing benign programs. While our clinic test could have a limited scope, we believe a well-designed clinic test is still helpful to refine our automatically generated vaccines in a real-world scenario.

4.5 Limitation

4.5.1 Complementary to Existing Malware Detection System

We consider malware vaccine as a complementary of the existing malware detection system. In particular, malware detection system normally assumes the existence of the malware at the end-host and its task is to find the most persuasive evidence to prove such existence. Whereas malware vaccine is used to break the *malware survive chain*.

More specifically, AutoVac makes a tentative step in deriving vaccines to defending malware. In general, malware vaccines aim to break these specific survive requirements. Such requirement can be illustrated in the following three scenarios:

- Some malware can only survive in the scenario when none of the same malware instance is present in the host. Thus, they have to uniquely *mark* their infected system. Our vaccine can hence appears to be uninfected with the infected system that can confuse the malware and stop its execution.

- Some malware has some programming fault in handling the failure of system resource access. Vaccine is trying to enforce such failures to make the malware go to their undesired status.
- Some targeted malware is designed to work in specific system environment. Vaccine attempts to make each protected system different from malware targeted environment.

4.5.2 Possible Evasions

It is possible to evade our vaccine if malware authors are aware that we are using certain resource as the vaccine. They can opt to drop the specific resource checking logic or change the resource name in the new version. However, the former will possibly leads to re-infection and thus may be not desired.

While the latter approach of changing resource name is possible, if we consider the wide and random propagation of worm or botnet malware, our vaccine still makes the malware hard to decide whether the system has actually been infected or not. Hence, if the malware binary cannot run with over two instances on same machine, our vaccine can bring the malware into a dilemma that the target system may actually been infected before or it has installed our vaccine system. Even though malware can run with multiple instances, periodically changing the identifiers may finally result in multiple instances running in one machine. It also create extra risk for being detected.

Certainly, malware author could also obfuscate the malware code to frustrate our vaccine generation such as using control dependence to propagate data. In fact, in many cases, there is actually no propagation chain and the conditional check is directly operated with the resource values. While future malware could deliberately introduce additional data propagation and obfuscate through control dependence, to

address such problem will be one of our future efforts as discussed below.

4.6 Related Works

In [24], Manuel et al. proposed an end-to-end approach to make end-hosts immune from fast-propagating worms through collaborative worm detection and self-certifying alerts. Packet Vaccine [90] followed this direction and derived the network signatures of malicious packets to be used at the network level to filter unwanted packets. Different from these previous work, AutoVac does not investigate the exploits nor vulnerabilities that malware targets, and instead it analyzes the system resource constraints of malware and attempts to extract effective vaccines to immunize a clean system from future malware infection.

4.7 Summary

We have presented AutoVac, a new complementary malware defense system that aims to extract possible malware vaccines from given malware samples. We believe it is an appealing complementary technique in defending malware threats.

For these uninfected machines, our generated vaccines can significantly reduce the risk of malware infection. However, at the other side, for infected machines, we need to detect them and remove the malware. In the next two chapters, we move our focus on these machines and propose our solution to detect them.

5. ANALYZING AND DETECTING P2P MALWARE

5.1 Introduction

While many early botnets use centralized C&C architecture, botmasters have realized its limitations and begun to use more advanced and robust peer-to-peer (P2P) architectures for C&C [37]. For example, several contemporary successful botnets such as Storm/Peacomm and Conficker have infected millions of computers and adopted P2P techniques in their C&C coordination [73, 94]. As stated in a recent report [36], the Kaspersky Security Network detected more than 2.5 million P2P malware incidents per month in March 2010, a high water mark reached for the first time in its monitoring history. A recent P2P botnet, Sality, is still alive as of the writing of this dissertation and becoming more complex [35]. In short, P2P malware is widely believed to be a promising direction for future malware [37, 75, 58].

Unfortunately, to date, there is relatively little research available on detecting this important threat. Network-level detection techniques have been proposed to perform clustering/correlation analysis to identify suspicious botnet infection/behavior patterns [67, 39, 41, 38] or to analyze the network traffic graph/structure to detect possible P2P botnets [42, 45, 58]. However, suspicious pattern identification may fail in front of traffic encryption, traffic randomization and timing pattern manipulation [74]. Structure/graph analysis can only detect P2P structure regardless whether the traffic is actually malicious or not, and it typically requires tremendous resources (e.g., global ISP-level view) for acceptable results (a conclusion also mentioned in [45]), making it a less attractive solution to Enterprise networks. In another direction, host-based detection techniques such as traditional signature-based approaches (e.g., anti-virus tools) and more recent behavior-based approaches (e.g., [48, 47]) have

also been proposed. However, due to the widely used advanced obfuscation/polymorphism [70] and the requirement of client-side installation, the solutions are not attractive for large scale P2P malware detection. Finally, it is worth noting that both host-based techniques and the above-mentioned network-based approaches have one common limitation because of their passive monitoring mechanism: they tend to be *slow* in terms of detection, e.g., they need to wait until some (or many) actual (suspicious/malicious) activities/communications occur to be able to detect the malware existence.

In this section, we focus on answering the following question: *is it possible to combine both the robustness of host-based approaches and the efficiency of network-based approaches to provide fast, reliable, and scalable detection of P2P malware?*

We believe that while P2P provides more flexible and robust coordination for the enemy, we can utilize the enemy's strength against him. A key insight is that P2P malware has to have built-in remotely-accessible/controllable mechanisms. That is, P2P malware has to open some port(s) for peer-to-peer communication, which is required for providing binary downloading services to new infected machines (i.e., egg downloading [39]), or for easier later access/control by remote attackers. If we can determine the port number(s) in use and further know the access/control conversation logic through that port (we refer to this information as Malware Control Birthmarks, or MCBs, as defined in detail later), we could uniquely identify that P2P malware.

Our key insight motivates us to design a novel two-phase detection framework: (i) first, we automatically extract MCB through *host-level dynamic malware analysis*; (ii) then, with the MCB information, we perform *network-level, active, informed probing* to identify infected machines. Thus, a P2P malware sample will expose itself if it opens specific port(s) or/and it responds in a predicted way to a specific probing

packet. It is worth noting that our new detection scheme applies in general to any malware that has MCBs, not just to P2P malware. For example, Trojan/backdoors also belong to the detection scope of this scheme and they are among the current most popular malware in the wild as shown in a recent Symantec Internet security threat report [20].

Our new design naturally bridges host-based dynamic binary analysis and network-based informed probing. Compared with existing solutions, it has several unique advantages. First, it is fast and active compared to existing passive detection mechanisms. Instead of waiting for actual attacks/control to happen, we can proactively detect the existence of malware. Second, it is very reliable in detecting the malware. While attackers can generate very different binaries for samples in a malware family, the underlying MCBs are still the same and they are typically unique for different malware families. This is because the attackers still want to control all the malware (in the same family) in the same way to make them easily manageable. The accuracy and robustness of using MCB in detection are comparable to traditional host-based approaches (they both use fine-grained binary analysis techniques), and it avoids a lot of network evasions. Finally, our approach is scalable to large network deployments. Since we only need one scanner for the whole network instead of installing detectors on every machine, the deployment, management, and MCB updating are relatively easy. It even provides the possibility of Internet-scale scanning/detection when necessary.

5.2 Problem Statement

In this section, we formally present the problem of P2P malware detection and its assumption.

5.2.1 Assumption

We assume that a captured malware binary P is available, and we analyze it in our host-based analysis phase without source code access. With the wide deployment of honeypots to collect malware samples, this is a very basic assumption for most malware analysis and defense research [49, 48, 13, 57, 93]. Furthermore, since most malware binaries are now protected against static analysis (e.g., using obfuscation/polymorphic techniques), we mainly employ dynamic analysis techniques in this work.*

Since we target P2P malware, without loss of generality, we assume the malware sample P contains two independent program logics:

- P_1, which opens a network service port ψ.
- P_2, which parses certain network request(s) ρ and generates response(s) η through the network port ψ.

We assume all binary samples within the same malware family/version share the same and unique P_1 and P_2. These two program logics provide a remotely accessible/controllable mechanism that we capture as the *birthmark* of the malware family, which we call *Malware Control Birthmark* (MCB).

More formally, a MCB can be defined as a pair:

$$<\text{Portprint}\{P_1, \psi\}, \text{ MCB probing } \rho \text{ and response } \eta >$$

Here *Portprint* denotes the service port(s) ψ used by the malware and the corresponding algorithm/logic P_1 to generate such port number(s). *MCB probing* denotes some well-constructed probing packet(s) ρ that trigger(s) the execution of malware control logic P_2 to reply with some (network observable) unique response(s) η.

*Note that combining static analysis will definitely improve our approach.

5.3 System Design

Overall, we illustrate the design of PeerPress in Figure5.1.

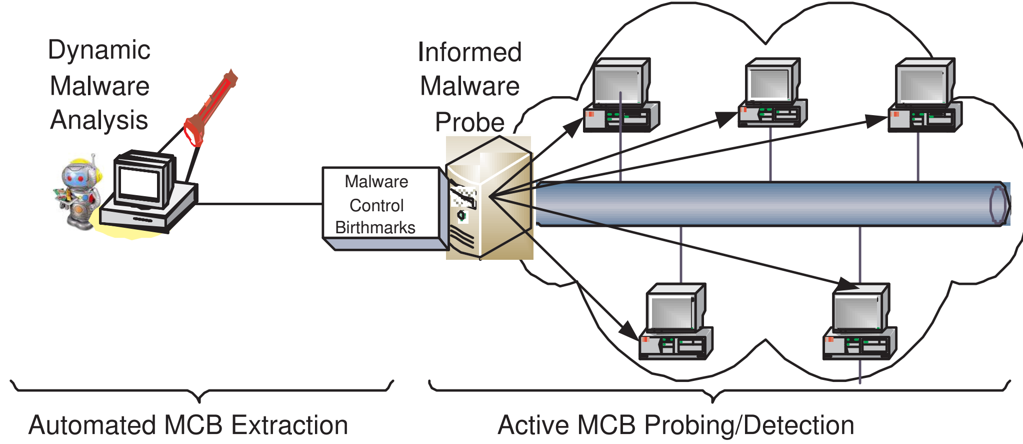

Figure 5.1: Our Two-phase Approach of PeerPress

5.3.1 Phase I: Malware Birthmark Extraction

The first phase is automated MCB extraction through dynamic malware analysis. In this phase, we analyze the malware sample and extract its MCBs (including both Portprint $\{P_1, \psi\}$ and MCB Probing $\rho, \eta)$ if possible.

- Portprint extraction. To identify a portprint, we first run the malware P in a test environment and collect the trace from the malware starting up to opening a socket and binding this socket to a port. We capture the network service port ψ and further reason about the generation of such port. If the port number is environment dependent and/or algorithmically generated, we need to further extract its generation logic P_1.

- MCB probing extraction. Using the same analysis environment, we begin with sending random fuzzing packets to trigger the execution of logic P_2. Leveraging the basic execution trace, we perform *directed, informed* multi-path exploration to identify interesting MCB execution paths. We further employ concrete and symbolic execution techniques to derive MCB probing packets (input) ρ and the corresponding response η. To verify the uniqueness of MCBs, we examine ρ and η to ensure it is not the similar benign traffic targeting port ψ.

Key Challenges and Solution. There are two main challenges we need to solve in PeerPress.

Challenge 1: Extracting and reasoning about the dynamic portprint $\{P_1, \psi\}$. It is worth noting that the port number ψ that we might observe in the analysis environment may not represent the actual port number that will be opened on compromised machines. This is because the malware instance P interacts with different environments on different machines, which could influence the generation of ψ. One real-world example is the Conficker worm [63], which binds to different ports based on different IP addresses. Although we know that P generates ψ in the analysis environment E_t, we still need to derive the corresponding port ψ_i in the probing environment E_i when infected by the same malware. The dynamic attribution of the listening port on targeted machines represents a challenge.

We find that malware generates its listening port in three ways:

- Static. In this case, the malware always opens a fixed port number, which might be defined in a configuration file or is embedded in the binary. For example, Nugache [75] always listens on TCP port 8.

- Algorithmically deterministic. In this case, the malware uses some algorithm to generate a host-specific port number. This algorithm can take various pa-

rameters, e.g., IP address and time. Conficker.C belongs to this type [94]. We envision that more future malware might use this advanced feature because it removes the need of some central servers or super peers to collect port information and then coordinate/distribute among other nodes for bootstraping peer discovery in traditional P2P malware.

- Random. The malware listens on some randomly generated port. In this case, our probing scanner will have to utilize existing network traffic monitoring or port scanners to identify the opened ports on end hosts. With the widely deployed network monitoring and scanning tools already available to network administrators, this should not be a significant issue.

Thus, an effective solution should tell us the portprint type of a given malware program (static, random, or algorithmically deterministic). Furthermore, it should provide the port generation logic/algorithm P_1 (particularly when it is algorithmically deterministic) and the knowledge of the environment it depends on (e.g., IP/Mac address, machine name, or system time). In this case, given a new target machine i to scan, we can run the same portprint logic P_1, simulating environment e_i on machine i as the input parameters to generate the target port.

The problem of determining the type of portprints and the sources of portprints can be solved by using well-known taint analysis techniques [71, 101, 89]. However, different from most traditional *forward* taint analysis work [101, 71] to solve known-sources-to-unknown-sinks problems, our problem is essentially *many-unknown-sources-to-one-known-sink*. Thus, we start from the port number and perform offline *backward taint analysis* to obtain the complete data dependence flow for the port generation. Based on the semantic meaning of the sources, we can determine the portprint type, and the necessary environment parameters that will contribute

to the port generation. Furthermore, to extract the portprint generation logic P_1 as an independent program, we apply classic backward program slicing techniques [54] in a similar way to related work [49, 48, 13].

Challenge 2: Efficiently Exploring and Extracting MCB Paths Inside P_2. Regarding the packet parsing logic P_2 inside P, we aim to find all possible execution paths that start from packet receiving routines (e.g., `recv()`) to packet transmitting routines (e.g, `send()`). This is a basic requirement for candidate MCB paths, because as we mentioned before, we assume a well-constructed MCB probing packet ρ can trigger a specific response η along a MCB path.

Thus, the problem becomes *how to efficiently find all possible MCB paths in P*? It seems that existing multipath exploration approaches [57, 11] could be applied directly. However, these approaches typically follow a depth-first search scheme and randomly choose a path when reaching any branch point. As a result, if they are used in our application, they will blindly explore all possible (although mostly unnecessary) paths to find desired MCB paths.

Our proposed solution, Informed enforCed Execution(ICE), combines both forced execution [93] and concrete/symbolic execution [89, 11, 57] techniques to improve the effectiveness and efficiency in finding MCB paths.

During execution, ICE first takes a breadth-first search approach to quickly obtain an overview on the packet processing procedure before going into any depth (sub-functions). Furthermore, ICE employs *directed* search when exploring paths at branch points with the intuition that some paths containing certain functions/calls are more likely MCB paths. Examples of these functions include those that directly call `send()`, or indirectly call functions that wrap `send()` (several layers of wrapping is possible here).

We use *function containers (FC)* to refer to such functions that when called they will reach our desired network routines such as `send()`. Code blocks leading to those *FC*s that end with valid network transmission such as `send()` are preferred when exploring paths. Moreover, a special type of *FC*s will denote functions that lead to network/process termination such as `closesocket()` and `exitprocess()` without sending out network information. Code blocks leading to these *FC*s should be given lower priorities. Basically, ICE automatically creates and maintains the list of different *FC*s and uses them to make the best possible decision at any branch point.

When exploring new paths at a branch point, ICE has a *Foreseeing* step to analyze the next k code blocks to decide the priority of branches to take. Generally speaking, ICE will prefer the branch containing high priority *FC*s and then force the execution towards that path.

5.3.2 Phase II: MCB-assisted Network Probing

The second phase is MCB-assisted network probing. We will use our extracted MCBs to guide probing of networked computers to quickly and reliably identify malware infected victims. More specifically, targeting the P_1-generated port ψ, we employ a network scanner S to probe each host. If we observe the desired ρ and η pair from probing, we report the machine as compromised (by the specific P2P malware).

5.4 Evaluation

5.4.1 Experiment Dataset

In this section, we evaluate PeerPress on several real-world malware families, which are listed in Table 5.1. This includes representative and complex modern P2P bots such as the infamous Nugache malware [75], Phatbot, Storm/Peacomm [73],

Conficker C [94], and more recent Sality [35](still active in the wild as the writing of this dissertation). We also include several Trojan horse/backoor malware, because they also contain MCBs (many of them could also be considered as bots). This is to further demonstrate that PeerPress can detect more than just P2P malware, as long as PeerPress can extract MCBs from the malware. These malware samples were collected from multiple online malware repositories such as[61, 3] and diverse security researchers. We verified the ground truth labels of these malware with multiple online malware analysis services such as [88, 3] and manual examination on binaries and network traffic.

Name	*Type*	*Name*	*Type*
Conficker C [94]	P2P Bot	Nugache [75]	P2P bot & Trojan Horse
Phabot [8]	P2P Bot	Sality [35]	P2P Bot
Storm/Peacomm [73]	P2P bot	BackOrfice	Trojan horse/backdoor
NuclearRAT	Trojan horse/Spyware	WinEggDrop	Keylogger/Spyware
Penumbra	Backdoor	WinCrash	Backdoor
NuCrypt	Trojan horse/worm	Wopla	Trojan horse

Table 5.1: 12 Malware Families in our Evaluation

5.4.2 Experiment on Effectiveness of Portprint Extraction

We extracted portprints for each malware family and we summarize them in Table 5.2. To verify their correctness, we run these malware multiple times in a clean environment and each time compare our extracted portprint with the actual port the malware bound to. The detailed result is shown in Table 5.2.

Among all the malware we have examined, Conficker C has a complex and unique port generation logic, which was previously manually analyzed in [63]. Now with PeerPress, we can automatically extract this logic within a few minutes.

Malware	*Type detmined by MProbe*	*Observed Port Number*	*Description*
Conficker C	algorithmically determined	46523/TCP and 18849/UDP	Program Slice with IP and time
Nugache	static/randomly generated	8/TCP, 3722/TCP	Open Multiple (fixed/random) Ports
Sality	algorithmically determined	6162/UDP	Generated based on Computer Name
Phabot	randomly generated	1999/TCP	
Peacomm	static	7871,11217/UDP	Read from spooldr.ini
BackOrfice	static	31337/TCP	In binary
NuclearRAT	static	190/TCP	In binary
WinEggDrop	static	12345/TCP	In binary
Penumbra	static	2046/TCP	In binary
NuCrypt	static	3133/TCP	In binary
Wopla	static	8080/TCP, 25099/TCP	In binary/file
WinCrash	static	1596/TCP	In binary

Table 5.2: Portprint Details of Different Malware Families

Furthermore, PeerPress provides a clear function interface with parameters and their semantic meanings because it captures system calls such as `getpeername` that parse the buffer related to the slice arguments. It is worth noting that algorithmically deterministic portprints are a strong evidence of the malware existence. That is, with only portprints (even without further MCB probing packets/response), we can already detect this kind of malware with very high confidence.

We find that many portprints are static in our tested malware. Most of such malware embeds the port number in the binary, such as NuclearRAT and NuCrypt, or reads from some configuration file, such as the case of Peacomm/Storm. Only a few malware samples (Phabot and Nugache) listen on totally random ports. In our tests, the ports were used for FTP services in both cases (to provide egg downloading service for newly infected malware). This inspired us to probe suspicious random ports just using an FTP packet and monitor their reply. In Section 5.4.4, we further demonstrate even though the malware may use the standard FTP protocol, the slight implementation differences may still expose themselves.

One very interesting case is the algorithmically deterministic portprint of Sality (UDP port), because previous reports have claimed that the port is selected pseudo-

randomly [35]. We carefully examine our generated portprint and find that there are two source bytes that are the result of system call `GetComputerName()`. These two bytes are multiplied, and the result is added to a constant number `0x438`. Meanwhile, through tracking the control dependences, PeerPress also successfully extracts another path which forces the malware to bind to a static port, 9674. We deduce that the reason why security reports such as [33] claim the port is psedo-randomly generated may be because: (1) The computer name can be considered as a random value. (2) It is possible for malware authors to reconfigure the constant number `0x438` to other constant value. PeerPress declares that the portprint of Sality is algorithmically deterministic, and it extracts the program slice with the target computer name as the parameter. Once provided with computer names (which should be available to most network administrators), PeerPress can probe target machines to detect Sality infected victims.

5.4.3 Experiment on Effectiveness of ICE

In this section, we evaluate the effectiveness of ICE. First, we conduct an experiment to verify that there are multiple function containers in each malware binary, which supports our assumption that function-level abstraction is feasible in dynamic analysis. Second, we verify that it can significantly reduce the overhead of path exploration compared to existing exploration scheme.

Function Containers in Malware Binary. In our evaluation, we set the maximum call depth level as 4, and locate on average 28 function containers per malware sample using this level.

In our tests, all containers eventually lead to desired system calls. More interestingly, throughout all our test cases, malware calls these containers if they want to execute specific tasks.

Overhead Comparison. To evaluate whether our informed execution can efficiently locate desired MCB logic, we compare the performance of ICE with the traditional approach that randomly chooses a path to explore next [93]. Here, the performance is measured using the *number of rounds* to find all MCB paths (that the system succeeds in finding using a brute-force approach), and each round is defined as one path exploration attempt from the sink (receiving the probing packet) to the end of the execution run for this path. Note that we do not claim to be able to explore *all* execution paths in the program. Instead, our baseline of all MCB paths is determined by brute-force exploration of all possible paths that can be directed/triggered by one single probing packet (i.e., we may miss MCB paths that can only be triggered by multiple probing packets). All these paths start from packet receiving till (i) the malware sends out some response, or (ii) the communication/process terminates. In this test, it is not very important whether we obtain accurately all MCB paths or not. Instead, more importantly we want to see which technique is quicker to locate these MCB paths given as the baseline. The result is shown in Figure 5.2. We can clearly see that our ICE significantly outperforms the traditional forced executions [93]. Our method requires much fewer exploration rounds to find MCB paths. In many cases, our system reduces the overhead up to 80%.

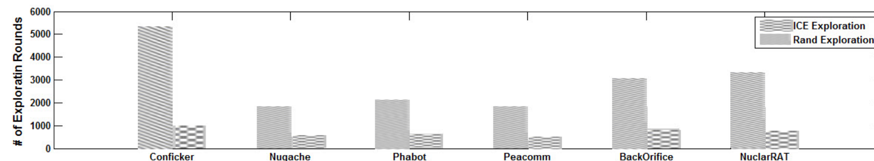

Figure 5.2: Performance Comparison of ICE and Random Exploration

5.4.4 Experiment on MCB Extraction

PeerPress successfully extracts on average about 6 MCB probing/response pairs per sample from all the tested malware, as shown in Table 5.4. In terms of running time, we select three most complex, representative malware samples and report the performance for different components of our system in Table 5.3 (performance of other samples are similar or better, omitted here due to space limitation). We acknowledge that some steps, such as semantic derivation and symbolic execution are relatively slow, which is not surprising considering that we are analyzing very complex real-world P2P malware in a fine-grained way with some known-expensive operations. Compared with existing state-of-the-art work (e.g., [14, 15]) that also uses expensive dynamic analysis and symbolic execution techniques, our performance is on par with those studies, and we believe it is reasonable and tolerable for offline analysis of malware families (recall that the analysis does not need to be repeated for each individual sample). It can certainly be improved by optimizing our code, parallelizing some operations, and using more powerful hardware.

	Conficker C	Nugache	Peacomm
Fine-grained Recording (min)	38	21	37
Backward Taint (sec)	243	549	780
Program Slicing (sec)	180	363	173
Semantic Derivation (sec)	2813	489	541
ICE engine (sec/trace)	54.4	38.9	40.3
Symbolic execution (sec/trace)	6863	1602	2711

Table 5.3: Running Time of MCB Extraction

Among all MCBs that PeerPress extracted, the simple case is represented by certain Trojan horses/backdoors that provide some unique "Welcome" information in their response. It is actually a very effective and safe MCB without much effort

Malware	*# MCB*	*Malware*	*# MCB*
Conficker C	3/3	Peacom	6/3
Sality	1/1	BackOrifice	16/14
Phabot	13/9	NuclearRAT	17/12
WinEggDrop	11/8	Penumbra	16/13
Nugache	21/7	WinCrash	1/1
NuCrypt	2/2	Wopla	2/2

Table 5.4: Statistics on Extracted Malware MCBs. (Here X/Y in Column *# MCB* means there are X candidate MCBs and Y final MCBs after verification.)

to generate. We can initiate connections to the suspected host and verify whether it welcomes us in the specific way or not. This welcome message is most common in old fashion Trojan horses, because an adversary may use any remote client to control the bots. We find this in Nugache FTP logic and some other malware, e.g., *WinCrash* and Wopla. For example, Nugache uses the following welcome message: `220-220-Welcome 220`.

We find that many MCB probing packets are easy to craft because there are no (or not many) encoding routines. More precisely, we found cleartext FTP logic inside Nugache, Wopla and Phatbot, peer synchronization logic inside Peacomm, and command and control logic inside traditional Trojan horses. Even though there are only a limited number of samples, our system is robust and fast to obtain MCB probing in a fully automatic way. In detail, we find one simple FTP service logic hosted by Nugache on a high-order port. After traversing the MCB paths, we extract 21 command and response pairs. After further verification, 14 are filtered (e.g., command `ls` and `pwd`) because they are not be considered as unique evidence.

As an interesting MCB example among the rest, we find that the Nugache FTP service needs users to provide `Username` and `Port` for validation, which are quite different from normal FTP services we see.

For Peacomm/Storm case, PeerPress extracted six MCB probing candidates.

We test these MCBs on the benign `eDonkey` clients and filter out three. One filtered example is a 509-byte probing packets (with the first two bytes as `0xe3 0x13`) that will receive a 18-byte response packet beginning with `0xe3 0x14`. This is actually used for regular peer recognition in the `eDonkey` protocol. The remaining three are interesting MCBs, include a probing packet beginning with the first two bytes `0xe3 0x0d` and the corresponding response packet beginning with `0xe3 0x0a`.

For the most sophisticated cases, we have to bypass the encoding function before the symbolic execution. As described before, we apply a semi-automatic approach to extract the encoding function inside of the traces. We automatically locate the `RC4` encryption and checksum routine inside Conficker and Sality, using several heuristics including highly-mixed receiving buffer [15]. We also successfully identify two double-word decryption keys inside the Conficker and Sality packet (with payload offset 2 and 0). Thus, we can recover the encrypted probing packet after the symbolic execution. Examining the cleartext payload, we find one key data field containing the payload version inside both Conficker and Sality. Both malware programs generate replies if the received payload version is lower or equal to its own binary version.

It seems that the P2P logic implements a self-updating procedure, and the only way to trigger its reply is to provide a payload with a suitable version number. Another interesting finding about the Sality botnet is the *double replies*. When we feed our probing packet, Sality sequentially replies with two packets. One packet attempts to start a new UDP session while the other one is a reply to our MCB probing.

Although PeerPress extracted MCBs from all tested malware, we note that it does *not* mean PeerPress can extract *all* MCBs inside malware. We actually encounter some issues due to some complex control logic inside some malware programs.

For example, PeerPress failed to extract MCBs from Nugache's port 8. We find multiple $WaitForSingleObject$ calls in the traces, waiting for some (asynchronous) event from other threads/process. ICE failed to correctly explore the paths in that situation. In the case of Conficker, PeerPress is not able to automatically crack the multi-round advanced encoding routines, thus failed to extract MCBs on some ports. The fact that PeerPress failed in several cases is not surprising, as we are dealing with real-world complex malware. However, our results are still encouraging because PeerPress could extract at least one meaningful MCB for all families that we examined.

5.4.5 Experiment on Detection Results through Probing

In this section, we conduct the experiments to verify that our MCB-informed active probing can detect our targeted malware in a *reliable*, *robust*, *fast* and *scalable* way.

Test in Virtual Networks. We built one virtual environment with six virtual machines. All virtual machines installed Windows XP SP1 without new patches. We randomly selected two different malware samples (from Table 5.1) to install on each machine (and eventually cover all twelve malware families in six VMs). Meanwhile, we installed some well-known benign services, such as Apache web server, P2P clients (e.g., edonkey), and FTP servers (e.g., Filezilla). Our probing engine uses extracted MCBs to actively probe the entire virtual network. PeerPress correctly detected all the existing malware in the virtual network without false positives. In terms of detection speed, it only took on average 1.103 seconds to detect each malware. This demonstrates that the informed active probing is an effective approach to detect malware in the network.

To further verify the robustness of PeerPress to detect different variants in the

same malware family, we further collected three additional (but different) binaries of the same malware for Conficker, Storm/Peacomm, and NuclearRAT, and Nugache, respectively[†]. Our test environment is the same as mentioned before.

PeerPress can not only detect all the variants but also correctly classified all variants into its original families. This again verifies that MCBs are unique for the same malware family and PeerPress is robust in detecting different malware variants in the same family.

False Positive Test in Real Networks. Next, we scanned our campus network (we randomly choose three /24 networks with no firewall to filter our scans) to test the real-world performance of PeerPress using the above extracted MCBs. We did not find any false positive during the scan, because most hosts do not have the corresponding (malware portprint specific) ports open. This is not surprising because our campus networks/computers are well managed/secured. We then intentionally scanned other open ports on these machines in order to further test the false positive of using MCB probing/response. We chose to scan port 80 (web) and all ports above 1025 in these three networks in hope to find some P2P applications. We found 58 hosts opened port 80 and 110 hosts opened higher ports, varying from several well-known P2P ports such as 6881 (BitTorrent) and 49153-49156 (uTorrent/Azureus) to some unknown ports. Our MCB-informed probing again did not yield any false positive. The probing speed for each host is about 1.128 seconds on average per MCB (including the first TCP port scanning interaction and the following MCB probing packet/response). Considering that it is easy to perform parallel scanning using multiple threads, PeerPress demonstrates good detection speed/scalability.

[†]For these four malware we could find different binaries/variants.

Comparisons with State-of-the-Art Detection Systems. In terms of an efficiency comparison with some state-of-the-art malware detection systems, we can mainly do a paper-and-pencil case study here because we could not obtain most of these tools. AccessMiner [52] is one relevant host-based detection system. It has a high accuracy and covers a lot of malware families. However, it may not be good enough at the stage where a P2P bot is waiting to receive commands from the botmaster, because it has not triggered its malicious logic yet. Meanwhile, it may also consume considerable resources on each end-host, so it is less scalable for deployment on large networks.

We further deploy another state-of-the-art network-based detection system, BotHunter [39], in our test (virtual) network and no malware (on six machines) is detected. This is reasonable because BotHunter needs to accumulate actual evidence related to multiple phases in the malware infection life cycle. In our cases, most of malware does not exhibit malicious network activity because the samples did not receive any commands. This also exposes one common limitation of many existing detection systems: they are passive and could be slow in terms of detection speed. On the contrary, PeerPress can actively detect those malware, even *before* those infected machine are accessed/controlled by remote peers/botmasters.

Note that compared with existing systems, PeerPress does have a limitation regarding to its detection scope. As clearly mentioned, PeerPress only targets malware that has MCBs, instead of all malware. However, we still consider it a valuable addition to our arsenal, because P2P malware and Trojan/backdoors are serious and emerging threats that we need to address. PeerPress greatly complements existing passive malware detection approaches.

5.5 Limitation

A notable limitation of PeerPress is that it cannot craft correct MCB probing packets in the case of advanced encryption or certificate-based authentication, even though it could identify/bypass these routines. However, this is a common problem for *all* malware analysis tools that aim to provide meaningful (network) input to malware samples [15].

Malware could use this to verify/authenticate our incorrect probing packets and refuse providing any future response. However, even in this worst case, we argue that this kind of "no response" is indeed a special, suspicious, recognizable response that could be used in MCB probing. Furthermore, we note that our technique can still successfully extract portprints, and in many cases, the portprint itself is enough to detect/confirm the malware (without actually sending MCB probing content).

To evade portprint extraction, malware authors may intentionally delay the port binding until some conditions are satisfied, e.g., the time reaches some specific date. Indeed, it prevents PeerPress from discovering the port binding at first sight with the cost of decreasing the utility (in terms of accessibility) of the malware. This issue could be solved if we skip all the `sleep()` related functions in the monitoring and analysis.

To slow down the analysis of ICE, malware authors may intentionally include many (bogus) branches directly after the packet receiving. Even in such case, ICE is still faster than random path explorations.

Another possible evasion is to faithfully mimic a benign normal protocol behavior. First, this will increase the workload of malware authors. Second, if not implemented faithfully, the malware still could be fingerprinted due to the subtle differences from normal protocols, as studies in this domain have shown [10, 16]. If the malware

authors choose to copy code from existing open source software in order to avoid differences in implementation, the code replication/copy [91] could become another possible point of detection.

Finally, we note that *within its detection scope* (when MCBs can be successfully extracted), PeerPress is fast, reliable, robust, and scalable. We believe it is a great complement to existing passive detection techniques even though it is not perfect (just as any intrusion/malware detection technique).

5.6 Related Works

We now review related works previously not mentioned.

5.6.1 Multiple-path Exploration

One related research is the exploration of dormant functionalities [19, 11, 57, 93] in malware binary. In [57], the authors take snapshots at each branch point and reset when an additional branch needs to be explored. Wilhelm et al. [93] present a forced sampled execution approach to explore multiple rootkit execution paths. However, both exploration schemes still depend on random choice because they cannot correctly define what is the target function they want to explore. Our goal is to explore the MCB paths, so the exploration can be effectively accelerated and the overhead is significantly reduced. Meanwhile, ICE solves the problem of exploring the sub-paths along one explored MCB main path, which is different from the problem solved by previous work.

5.6.2 Protocol Reverse Engineering

Automatic protocol reverse engineering (PRE) research [14, 26] discovers the semantic meanings of network protocols. However, these studies were mostly focused on analyzing legitimate network protocols. In such cases, it is easy to elicit a response

from the application, simply by using a legitimate client that sends a valid request. We do not know how a valid request looks like; in fact, one key aspect of our work is to efficiently locate MCB execution paths, which determine the format of probe packets that can be used to obtain responses. Moreover, PRE systems are broader in the sense that they attempt to reverse engineer entire packet formats and state machines. This is fine for legitimate applications, but might be too brittle when applied to malicious binary code. Our technique, on the other hand, focuses on a specific problem (the extraction of inputs that trigger responses), and hence, can be more robust. In addition, we introduce the idea of dynamic portprints, a concept that is not considered by PRE systems.

5.7 Summary

P2P malware is an important direction for future malware. Current P2P malware detection remains insufficient. In this chapter, we propose a novel, two-phase detection framework that seamlessly bridges host-level dynamic binary analysis and network-level informed active probing techniques. It can detect P2P malware and beyond, as long as the malware has MCBs.

As a similar communication mechanism, server-to-client is a more commonly applied in new malware attacks. Even some existing works can effectively detect malicious client by host-based detection scheme, detecting malicious server is still a big headache for defenders. In the next chapter, we extend our work to propose solutions for detecting malicious servers.

6. DETECTING INTERNET-WIDE MALICIOUS SERVERS

6.1 Introduction

Nowadays Internet is an essential part of our life. However, malware poses a serious threat to Internet security. Millions of computers have been compromised by various malware families, and they are used to launch all kinds of attacks and illicit activities such as spam, clickfraud, DDoS attacks, and information theft. Such malicious activities are normally initiated, managed, facilitated, and coordinated through remotely accessible servers, such as exploit servers for malware's distribution through drive-by downloads, C&C servers for malware's command and control, redirection servers for anonymity, and payment servers for monetization. These malicious servers act as the critical infrastructure for cybercrime operations and are a core component of malware underground economy. Undoubtedly, identifying malware's server infrastructure is of vital importance for defending against cybercrime.

Traditional approaches for detecting malicious servers mostly rely on passive monitoring of host and network behaviors in home/enterprise/ISP networks. However, such passive approaches are typically slow, incomplete and inefficient because miscreants use dynamic infrastructures and frequently move their servers (e.g., to be more evasive or as a reaction to takedowns). To solve this issue, active probing techniques have been proposed to detect malicious servers and compromised hosts in an active, fast, and efficient way [59, 98]. The basic idea is to send specially crafted packets (i.e., *probes*) to remote hosts and examines their responses to determine whether they are malicious or not. Since probes are sent from a small set of scanner hosts, active probing is scalable, even for the entire Internet.

We describe AUTOPROBE, which implements a novel approach to the problem

of automatically building fingerprints that can be used for (actively) detecting malware C&C servers on the Internet. Our goal is similar to the recently proposed CYBERPROBE [59], which demonstrated how active probing can successfully detect malicious servers at Internet scale. However, AUTOPROBE addresses fundamental limitations in CYBERPROBE.

First, CYBERPROBE is not able to generate fingerprints for many malware families that contain replay protection. In addition, the lack of semantics available in network traffic and the noise in the input network traces limits the quality of CYBERPROBE's fingerprints. Furthermore, CYBERPROBE cannot generate fingerprints when there is no known live C&C server to experiment with (thus no network interactions can be observed) or when the known C&C servers are only alive for a very short time (thus not enough traffic for building reliable fingerprints).

Dynamic binary analysis has been previously used by PEERPRESS to generate fingerprints for P2P malware. However, PEERPRESS cannot be used to detect remote malicious servers. It can only generate fingerprints for malware that embeds some server-side logic and listens on the network for incoming requests such as P2P bots. Instead, the majority of malware families use a pull-based C&C protocol, where bots contain only client-side logic, send periodic requests for instructions to the remote C&C servers, and close the communication after the response from the C&C server is received. Pull-based C&C is the dominant choice because it avoids incoming probes being blocked by NAT gateways and firewalls.

To build fingerprints for remote servers PEERPRESS would require the C&C server software, which is not available. AUTOPROBE is the perfect complement to PEERPRESS. Using both systems fingerprints can be generated for malware that uses both push-based and pull-based C&C protocols.

AUTOPROBE generates fingerprints for probing remote servers for malware that

has only client-side logic. AUTOPROBE applies dynamic binary analysis to achieve better understanding on the semantics of packets and deeper insight on malware's logic of generating requests (to remote servers) and handling responses (back from remote servers) in the following ways.

- In analyzing (outgoing) request generation logic, AUTOPROBE focuses on two tasks: (1) It tracks the generation of variant bytes, whose value may change in a different environment, and their semantics. Through re-generating variant bytes in realistic environments, AUTOPROBE attempts to obtain a more accurate probe request. (2) It analyzes the logic to uncover as many request generation paths as possible. Thus, AUTOPROBE can generate more probing requests that cannot be observed in the normal run by existing approaches.

- In analyzing (incoming) response handling logic, AUTOPROBE employs a novel scheme for detection, i.e., AUTOPROBE identifies specific response bytes that can affect client-side malware's execution as the evidence to detect malicious servers. More specifically, AUTOPROBE applies dynamic symbolic execution to find a set of path constraints and generate light-weight network-level symbolic-constraint-based fingerprints for detection evidence. Furthermore, AUTOPROBE provides practical solutions for handling real world challenges, e.g., when a remote server is not alive thus no actual response can be received by the malware client, which cannot be handled by existing approaches such as CYBERPROBE.

6.2 Problem Statement

Active probing (or network fingerprinting) is a powerful approach for classifying hosts that listen for incoming requests on the network into a set of pre-defined classes based on the networking software they run. In a nutshell, active probing sends a

probe to each host in a set of targets, and applies a classification function on the responses from each of those target hosts, assigning a class to each host. Given some target network software to detect, a *fingerprint* captures how to build the probe to be sent, how to choose the destination port to send the probe to, and how to classify the target host based on its response.

The problem of active probing comprises two steps: *fingerprint generation* and *scanning*. We focus on the fingerprint generation step, proposing a novel approach to automatically build fingerprints for detecting malware C&C servers. Our approach assumes the availability of a malware sample and applies dynamic binary analysis on the malware to build the fingerprint.

6.2.1 Advantages of Binary-Based Fingerprint Generation

A program analysis approach to fingerprint/probe generation addresses the following challenges that network-based approaches suffer.

Produces Valid C&C Probes. Network-based approaches produce candidate probes and send them to the remote servers to observe whether any of the candidate probes incites a distinctive response. Those candidate probes can be manually selected using protocol domain knowledge [16], generated randomly [16], or selected from prior messages the malware has been observed to send [59].

However, these three approaches are problematic. First, domain knowledge is not available for most C&C protocols. Second, randomly generated probes are most likely invalid because they do not satisfy the C&C protocol syntax or semantics. A remote C&C server is likely to refuse responding to invalid probes and the malware owners may be alerted by the invalid requests.

Third, previously observed malware requests may be invalid when replayed at a different instance of time and by a different machine from the one that originated

them. For example, a C&C request could include some replay protection such as a timestamp of the time when it is sent or a nonce previously received from the C&C server. When replaying a probe at a future time the probe violates the semantics of the timestamp and nonce fields. Similarly, a C&C request may contain fields that need to be filled with the IP address or OS version of the infected host. Replaying such requests on a different machine may produce semantically invalid probes. Those inconsistencies may be detected by the C&C server, which may refuse to respond to the probe and thus it may not be possible to build a fingerprint for the malware family. Encrypted or obfuscated C&C protocols make the problem even harder for network-based approaches as the protocol semantics are hidden behind the obfuscation layer, but revealed during the program's execution.

Explores the Space of Valid C&C Probes. CYBERPROBE is limited to use probe requests that have previously observed being sent by the malware. However, those requests are often only a small subset of all probes the malware can generate. For example, a malware may use the following probe generation logic: "sprintf(url, "/ark%d", rand() % 10000). While some such URLs may have been observed on network traffic, it is highly unlikely that all 10,000 possible URLs would have been observed. AUTOPROBE is able to extract the above request generation logic from the malware and thus can produce all 10,000 possible requests. The use of different probes during scanning makes the overall scanning less noisy, producing fewer complaints.

Minimizes False Positives. One goal of *adversarial fingerprint generation* is to minimize the amount of traffic that needs to be sent to remote C&C servers during fingerprint generation. As a consequence, few responses are available to build a signature on the response (as CYBERPROBE does). When faced with insufficient

training data, machine learning approaches can introduce false positives. Instead, AUTOPROBE leverages the intuition that the malware that produces the request knows how to check if the received response is valid.

By examining the malware's request handling logic AUTOPROBE identifies the checks the malware performs to determine if the response is valid, which AUTOPROBE uses as a signature that minimizes false positives.

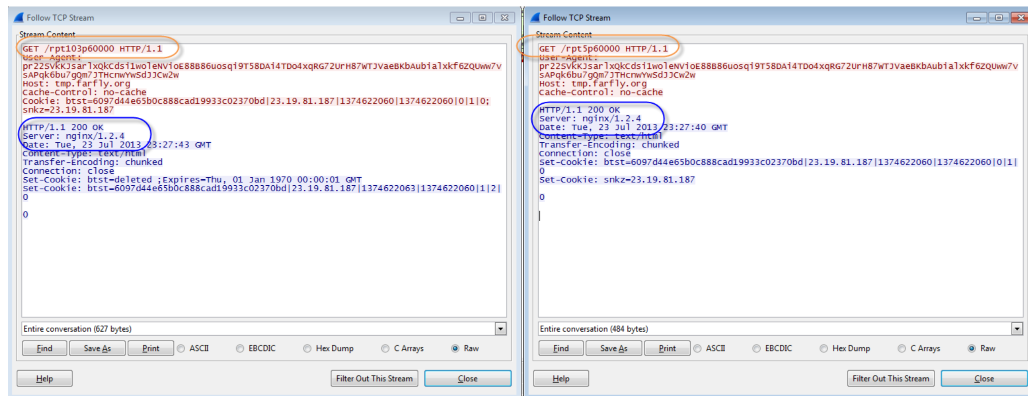

Figure 6.1: Two Network Requests Produced by Win32/Farfli.C

Does not Require a Live C&C Server. Network-based approaches to fingerprint generation [16, 59] assume that at least one request-response interaction between malware and a C&C server has been captured on a network trace. However, an analyst often only has a malware sample that when executed no longer successfully connects to a live C&C server. That does not mean the operation to which the malware belongs no longer exists. Most often, the malware sample is simply old and tries to connect to dead C&C servers that have since been replaced with fresh ones. AUTOPROBE is able to generate fingerprints even when there is no known live C&C server from the malware family of interest to experiment with. The produced

fingerprints can be used to scan for fresh servers that may have replaced the old ones.

```
1  int a = input("%d")? input("%d"): EA60 ;
2  int b = 0;
3  sprintf(url_str, "GET /rpt%dp%d", 5, a);
4  if (InternetOpenUrl(handle, url_str) == VALID) {
5      if (CreateFile(NAME1) == SUCCESS)
6          b += 1H
7      if (CreateFile(NAME2) == SUCCESS)
8          b += 2H
9      if (CreateFile(NAME3) == SUCCESS)
10         b += 64H
11     sprintf(url_str, "GET /rpt%dp%d", b, a);
12     if (InternetOpenUrl(handle, url_str) == VALID)
13         InternetCloseHandle(handle);
14 }
```

Figure 6.2: Request Generation Logic of Win32/Farfli.C

Example. We illustrate some limitations of network-based fingerprint generation using a simplified example from *Win32/Farfli.C*. Figure 6.1 shows 2 HTTP GET requests generated by a sample of *Win32/Farfli.C* when it is executed. The first request is for URL "/rpt5p60000" and the second for "/rpt103p60000". On the network these two requests look very similar and may be clustered together. However, the malware's request generation logic is shown as pseudo code in Figure 6.2. It shows that both requests are generated by different logic and that there is a dynamic parameter in both requests that is always "5" in the first request, but a variable number on the second request, whose value depends on whether the malware was able to create three different files.

Overall, the logic shows that the `b` variable can have 9 possible values and a total of 10 different request can be produced by the malware.

This example motivates how the semantics of the malware request may be complex and nearly impossible to decipher from network traffic and how understanding

the malware's request generation logic enables us to identify the complete space of request the malware may produce.

6.2.2 Problem Definition

We address the problem of *automatic fingerprint generation*. Given a malware sample P from a malware family F the goal of automatic fingerprint generation is to automatically produce a *fingerprint* ϕ that can be used to scan for malicious servers belonging to family F located somewhere on the Internet. We assume the server-side code is not available in any form. The malware sample is provided in binary form with no source code or debugging symbols. We assume the malware sample initiates a set of requests S to contact its malicious servers.

A fingerprint comprises three elements: a *port selection function*, a *probe generation function*, and a *classification function*. AUTOPROBE builds these 3 functions using dynamic binary analysis on the malware sample.

The malware may select to which port to send a probe based on its local environment and the C&C server to be contacted, e.g., based on the time when the probe is sent and the C&C's IP address. Thus, the port selection function takes as input the local environment of the scanner host where it is executed and the target address to be probed. It returns the TCP or UDP port to which the probe should be sent.

The probe generation function takes as input the local environment and the target address to be probed and outputs the payload of the probe to be sent to the target address. Building the probe generation function comprises two steps:

- Identify the variant and invariant fields of each request r the malware sends.
- For each variant field, generate a re-generation logic which determines the value of the field based on the local environment of the scanner host and the target's address.

The classification function is a boolean function that takes as input the response from a target server, the local environment, and the target's IP addresses. It outputs true if the received response satisifies the checks that the malware performs on the response, which means that the target server belongs to family F. If it outputs false the target server does not belong to family F. We assume the malware sample performs checks on the response to determine that the response is valid, i.e., to determine that it understands the response. Otherwise, the probe is discarded as its response does not allow to classify target servers with certainty and would introduce false positives.

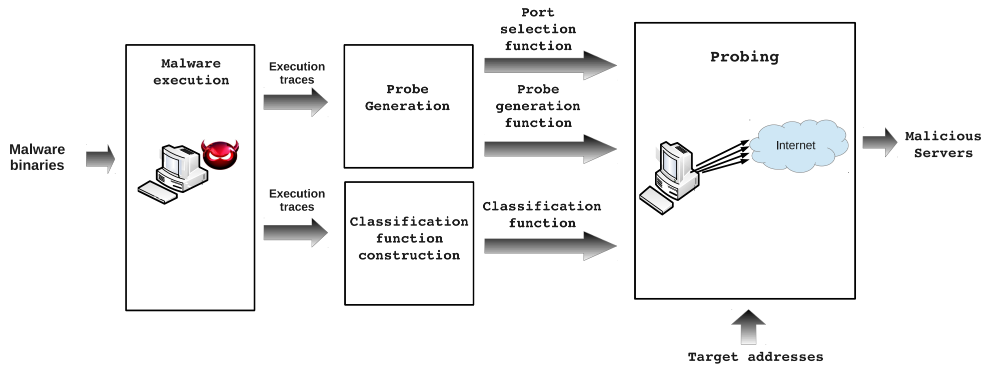

Figure 6.3: System Architecture of AUTOPROBE

The classification function is a conjunction of boolen expression that correspond to validation checks that the malware performs on a received response. It can be expressed directly on the raw byte string or on the protocol fields if the malware uses a known C&C protocol like HTTP. In the latter case it is used in combination with a protocol parser. An example classification function is shown in Figure 6.4. The malware checks that the response is successful (200 status code), that there is an HTTP body, and that the HTTP body contains one of three command strings.

```
1   if(InternetOpenUrl(handle, url_str) == VALID) {
2     if(!HttpQueryInfo(handle, HTTP_QUERY_STATUS_CODE,
          &status)) {
3         if (status != HTTP_STATUS_OK)
4           return ERROR;
5     }
6     if(!HttpQueryInfo(handle, HTTP_QUERY_CONTENT_LENGTH,
          &length))
7         return ERROR;
8     while(length) {
9         InternetReadFile(handle, lpBuffer, &bytes);
10        sscanf(lpBuffer, "<a>%d</a>", &command);
11        if (command <= 3 && command > 0) {
12            ... //
13        }
14        length -= bytes;
15    }
16  }
```

```
S1 = get_from_header(STATUS_CODE)
S2 = get_from_header(LENGTH_CODE)
S3 = get_payload()

S1 == 200 &           // Status code is 200
S2 >= 0   &           // Response has payload
(SEARCH(S3, "<a>1</a>") |
 SEARCH(S3, "<a>2</a>") |
 SEARCH(S3, "<a>3</a>") ) // Contains string
```

Figure 6.4: Classification Function Example

6.3 System Desgin

Figure 6.3 shows the architecture of Autoprobe. It comprises 4 phases: *malware execution*, *probe generation*, *classification function construction*, and *probing*.

6.3.1 Phase I: Malware Execution and Monitoring

Autoprobe first runs the malware executable inside an execution monitor that introspects the execution, monitors the system and API calls the malware uses, and produces an instruction-level trace of the execution. The execution monitor is implemented at the hypervisor-level so that the malware executing in the guest OS cannot interfere with it. The execution monitor is located inside a contained network environment that proxies communications to the Internet. The DNS proxy forwards DNS requests from the malware to the Internet. To incite the malware to start a C&C connection, if the DNS resolution fails, the DNS proxy creates a dummy response that points to a sinkhole server. For other TCP and UDP traffic Autoprobe uses whitelists to determine if the connection is considered benign and should not be analyzed (e.g., connection to top Alexa sites used by malware to check for connectivity) or if it is a C&C connection.

6.3.2 Phase II: Probe Generation

The goals of probe generation are to produce a probe generation function that captures the valid C&C requests the malware may generate based on its environment, and a port selection function that captures the port where the request should be sent. It applies dynamic binary analysis on the execution traces collected from the malware's execution. When needed, it invokes the malware execution component to obtain further execution traces. Figure 6.5 illustrates the architecture of the probe generation phase, which comprises 2 main steps: *exploration* and *trace analysis*. The exploration component executes multiple paths in the malware's request generation logic to identify different requests the malware may generate. The trace analysis component identifies the variant parts of a request, identifies their semantics, and produces regeneration slices for them. These two steps output the port selection function and a clasification function that captures the valid requests the malware may generate.

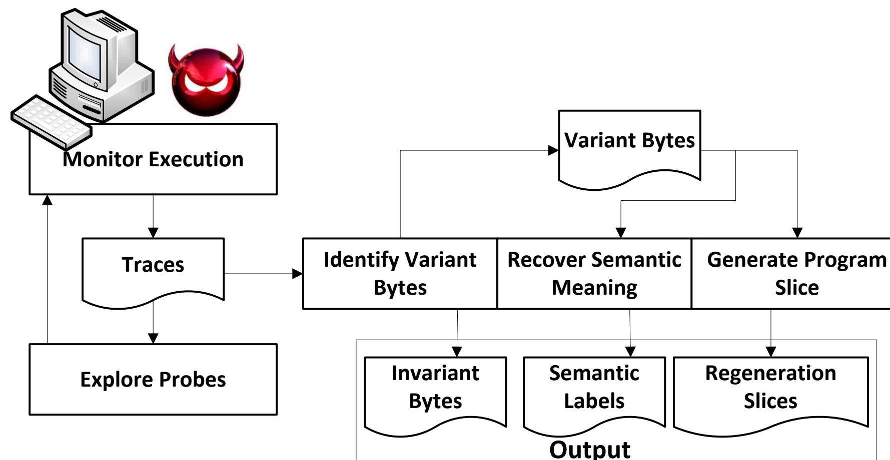

Figure 6.5: Probe Generation Architecture

Exploration. As mentioned, one limitation of dynamic analysis is that it only analyzes one execution path in the malware's request generation logic. The analyis of a single execution typically captures a large number of different requests that the malware can generate by modifying the values of variants fields in a request. However, it cannot capture different requests that the malware may generate depending on control-flow decisions on the running environment, i.e., on the output of system calls.

```
1  int rand_num = sub_0343(time);
2  sprintf(url, "/v1.0.1/?v=3.0&c=%ld", rand_num);
3  if (RegOpenKeyEx(PARA_KEY_PATH)) {
4     RegQueryValue(PARA_KEY_PATH, NULL, data, NULL);
5     sprintf(para, "PARAM=%s", data);
6     strcat (url,para);
7  }
8  if (InternetOpenUrl(handle, url) == VALID) {
9     // Handle Response.
10 }
```

Figure 6.6: Network Request Generation logic of Win32/LoadMoney.AF.

Figure 6.6 illustrates this problem. The malware checks the existance of a registry key using the `RegOpenKeyEx` function (line 3). If the call fails, the HTTP GET request sent by the malware contains a URL formatted according to line 2. But, if the call succeeds, the malware modifies the URL format by appending an additional parameter value to the end of the URL (lines 5-6). To understand that the malware can produce two different types of requests AUTOPROBE needs to explore the two execution paths introduced by the branch at line 3. For this, AUTOPROBE uses exploration, a technique that modifies the output of system calls that influence the

request generation logic.

Exploration is described in Algorithm 2. It performs a backwards analysis on the execution trace starting at the function that sends the request, e.g., `InternetOpenUrl` on line 8 in Figure 6.6. For each branch it encounters, it performs a backward taint analysis on the `CFLAG` register to check if the `CFLAGS` has been influenced by the output of a system call. If it is not influenced then it keeps processing upwards until it finds the next branch. When it finds a branch that has been influenced by the output of a system call (line 3)* it forces the system call to generate an alternative result. In our example, if in the original trace `RegOpenKeyEx` returned `SUCCESS`, it forces the function to return `FAILURE` so that the other execution branch is executed. This process stops when the beginning of the execution is reached or a configurable maximum number of system-call-influenced branches has been found.

Algorithm 3: Algorithm for Control-flow-based Exploration

```
Θ: Trace
ins: instruction in trace
Φ: Set of Instruction of Conditional Branches
Δ: Set of Labeled System Call Output Memory/Register
T: Set of Tainted Memory/Register
F: Set of System Calls Affecting Control Flow
req: Request Sent by Malware
for ins_i in Θ do
    if ins_i in Φ then
        eflags → T
        Backward Taint eflags
        if tainted ∈ Δ then
            Record System Call into F
            Clean eflags
        end
    end
end
for fun in F do
    for output:o_i of fun's outputs do
        if o_i changes control flow then
            Rerun malware
            Enforce o_i for fun along execution
            Collect new trace Θ_i Collect new req_i
        end
    end
end
```

*Or an API call known to perform a system call like RegOpenKeyEx

Trace Analysis. The analysis of an execution trace that produced a network request comprises 3 steps: identify the variant bytes in the request and the target port, recover the semantics of variant bytes in the request, and generate a regeneration slice for the variant bytes in the request and the port.

The request is commonly a combination of invariant and variant bytes. To identify variant bytes in the request AUTOPROBE applies dynamic slicing to each of the bytes in the request starting from the function that sends the request. Note that while each byte slice is independent they can be performed in parallel on a single backwards pass on the trace for efficiency. If the slice ends in a fixed constant such as an inmediate value or a constant in the data section then the byte is considered invariant. If the slice ends in the output of an API call with known semantics and whose output is influenced by a system call (e.g., `rand`), it is considered variant. In this case, AUTOPROBE clusters consecutive bytes influenced by the same API call (e.g., all consecutive bytes in the request influenced by `rand()`) into variant fields. Then it labels those variant fields using the semantic information on the API call collected from public repositories (e.g., MSDN). Some examples of semantic labels are *time*, *ip*, *random*, and *OS version*. Overall, AUTOPROBE has semantics information for over 200 Windows system and library calls. The handling of the port selection is similar but it starts at the function that selects the port (e.g., `connect`, `sendto`) and since the port is an integer value, AUTOPROBE can slice for all bytes that form the integer simultaneously.

For each variant field in the request the probe construction function needs to capture how the variant field needs to be updated as a function of the scanner's environment (e.g., the current time). For this, AUTOPROBE applies dynamic slicing on the previously identified variant bytes. The slice contains both data and control dependencies. For control dependencies AUTOPROBE conservatively includes in the

slice the `eflags` register value for each branch instruction it encounters that may influence the generation of the variant bytes. The slice ends when all variant bytes are traced back to some semantic-known system calls or the trace start is reached. The slice constitutes a program that can be re-executed using the current local environment (e.g., local IP, MAC address, or time) to reconstruct the field value.

6.3.3 Phase III: Classification Function Construction

To build the classification function, AUTOPROBE conducts dynamic binary analysis on the malware's response handling. Figure 6.7 depicts the architecture of the classification function construction. The intuition behind this phase is that the malware's processing of a reponse typically comprises two widely different logics to handle valid and invalid responses. For example, if the response is considered valid, the malware may continue its communication with the remote C&C server, but if considered invalid it may close the communication or re-send the previous request. To verify the validity of a response the malware parses it and checks the values of some selected fields. Such validation checks are branches that depend on the content of the response. Each check can be captured as a symbolic formula and their conjunction can be used as a classification function.

With a C&C Response. To differentiate valid and invalid responses AUTOPROBE focuses on the differences between validation checks on invalid and valid responses. For example, a valid response will successfully go through all validation checks but an invalid response will fail at least one of those checks producing an execution trace with a smaller number of content-dependent branches.

This case comprises 3 steps shown in the left part of Figure 6.7. First, AUTOPROBE marks as symbolic each byte in the response received from the server during the original malware execution and performs symbolic execution on those symbols

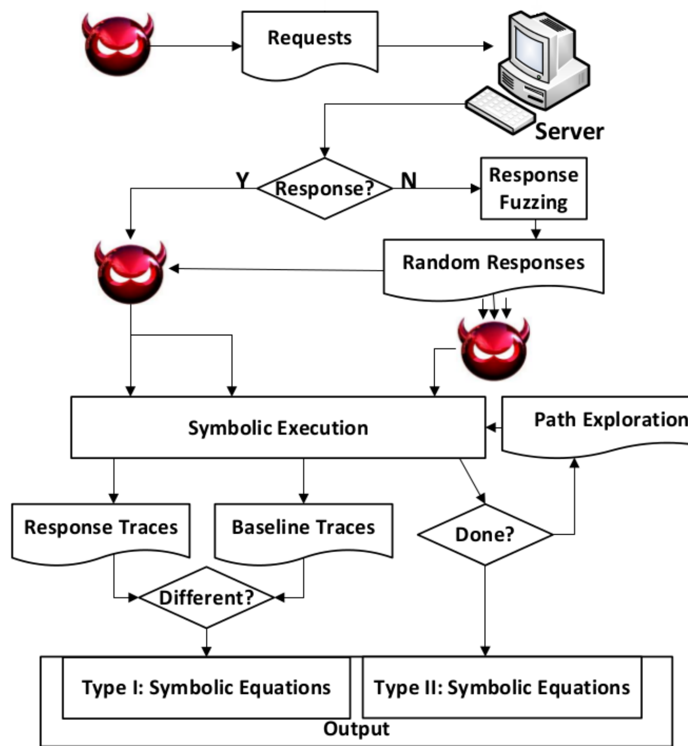

Figure 6.7: Classification Function Construction

along the execution. For each branch influenced by the input symbols (i.e., validation check), it produces a symbolic expression that summarizes the check. The symbolic execution stops when execution reaches some preselected calls such as `closesocket` and `exitprocess`, or when no validation check is found in the previous 50 branches. In addition to the symbolic formula, AUTOPROBE also outputs a θ_1 forward slice containing all instructions that operate on symbolic inputs.

Second, AUTOPROBE repeats the previous step but this time on a randomly generated (i.e., invalid) response. If the C&C base protocol is known (e.g., HTTP) rather than a random response AUTOPROBE uses a generic error message (e.g., an HTTP 404 response). The outcome is another symbolic expression and a θ_2 forward slice.

Third, AUTOPROBE determines if the θ_1 and θ_2 slices capture the same logic or not. For this, it aligns them and produces a δ slice, which records the instruction differences. Then it computes the distance between both slices η as:

$$\eta = \frac{\theta_1}{\theta_2} = \frac{\omega_{bn}\Sigma_{bn_1} + \omega_{fn}\Sigma_{fn_1}}{\omega_{bn}\Sigma_{bn_2} + \omega_{fn}\Sigma_{fn_2}}$$

where bn and fn are respectively the number of unique code blocks and unique system calls in δ. The weight values are set as $\omega_{bn} = 0.4, \omega_{fn} = 0.6$ to give higher preference to unique systems calls, which better capture malicious behaviors.

If η is below threshold m (experimentally set to 10), the response is discarded since it is handled similarly to the random response and thus is likely invalid. Otherwise, AUTOPROBE considers both executions different and extracts the symbolic execution results as two sets of equations, S_t and S_n, representing the validation checks results for valid and invalid responses.

During probing, AUTOPROBE compares the response from a target server with these two sets of symbolic equations. It determines that the target server is malicious if the response satisfies all symbolic expressions in S_t and none of the symbolic expressions in S_n.

Without a C&C Response. In this scenario the malware did not receive a response from any C&C server during malware execution. To address this case AUTOPROBE uses the approach illustrated on the right part of Figure 6.5, which comprises two steps: *fuzzing responses* and *exposing possible malicious logic.*

The first step is to fuzz the malware with multiple responses. When the C&C protocol is unknown the fuzzing uses random responses. If the C&C base protocol is known (e.g., HTTP) it starts with a successful response such as 200 OK and then continues with other valid message types.

For each pair of responses AUTOPROBE calculates the distance (η and finds the pair with the largest η.

Algorithm 4: Informed Enforced Execution in AUTOPROBE

```
Θ: Execution Trace Execution
Θ0: Execution Trace For Random Response
P: Malicious Program
pc: Instruction Pointer
S: Set of Symbolized Set for Response
Φ: Set of Branches Instruction
Ψ: Output Symbolic Equations Set
Symbolize all bytes in Response
Running Malware P
for eip do
    Enable Forward Symbolic Execution if eip ∈ Φ then
        if eflags symbolized then
            Save Execution Snapshot i
            Enable Enforced Execution
            Revert eflags
            Disable Enforced Execution
            Monitor Execution and Collect Θi
            Calculate ηi
            if ηi ¿ η0 then
                Online solving symbols
                if Solvable then
                    Save Trace Θi
                    Add Symbolic Equations for Θi to Ψ
                end
                else
                    Recover to Snapshot i to eip
                    Continue Execution
                end
            end
        end
    end
end
```

From this point that exposes most malicious behaviors, AUTOPROBE starts the exploration. AUTOPROBE conducts informed forced execution on all response-sensitive branches. Forced execution is a binary analysis technique which *enforces* the program to execute specific path and expose more behaviors of binary. However, there are two limitations for forced execution. First, the enforced execution may not be *reachable* in the real execution environment because the condition cannot be satisfied. Second, brute-forced exploration is tedious and inefficient. It requires a guideline for

enforcing more meaningful paths. To solve these issues, we combine symbolic execution with forced execution for our task. The algorithms is shown in Algorithm 4. In particular, we symbolize each byte in the response and continue online symbolic execution. If we find any branch's decision is dependent on the symbolized byte, we record the branch. Then we enforce the branch go to the unexplored path. Next, we calculate the η value of original path and explored path, if η value is increasing, we record the symbolic equation for the altered path. We iteratively continue the exploration and find all symbolic equations which increase η value.

6.3.4 Phase IV: Probing

The probing phase takes as input the target IP ranges to probe (e.g., the currently advertised BGP ranges) and the fingerprint. It uses the port selection and probe generation functions to send the probe to a target, and applies the classification function on the response, determining if each target is a server of the malware family of interest.

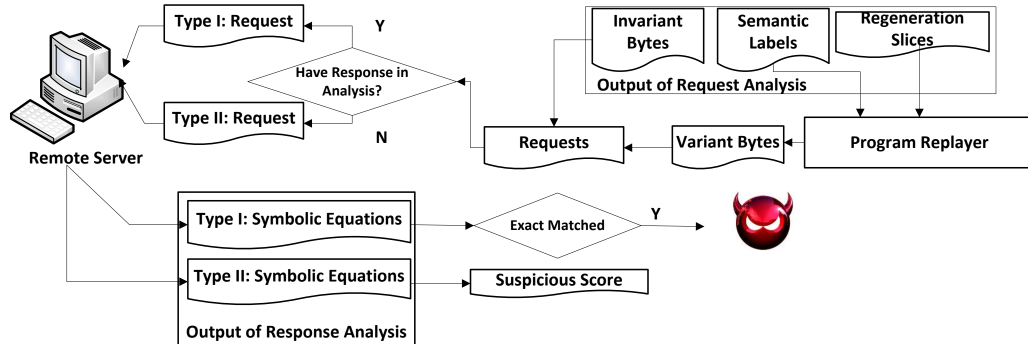

Figure 6.8: Probing Procedure of AUTOPROBE

In detail, the outcome of malware analysis consists of two parts. In the first part, we have a request string. In the request, if we find any string needs to be dynamically

re-generated, we fetch the program slice and re-generate the request. After we obtain the whole request, we send the request to our target probing machine and record the response. Then the response will be matched with a series of symbolic equations corresponding to the request. If the request is generated from our non-response analysis, the detection result is a suspicious score,

$$\lambda = \frac{\#\ of\ matched\ equations}{\#\ of\ equations}$$

The higher λ the more likely the target server is malicious. Otherwise, if the request is generated from concrete (live) server's response, we require the response satisfy all the symbolic equations to declare the detection.

6.4 Evaluation

In this section, we first evaluate AUTOPROBE for generating fingerprints of real-world malware samples. Then, we use the fingerprints to scan for malicious servers.

6.4.1 Experiment Dataset

We collect recent malware from 24 families broken into two datasets. Dataset I contains 100 malware samples from 10 popular malware families (10 samples each) that we collect from Offensive Computing, a public malware repository [61]. This dataset includes notorious malware such as Sality [33], ZeroAccess [96], Ramnit [65], Bamital [6], Taidoor [78], Mebroot [56], W32.Xpaj.B [97], and Amonetize.Q [2]. Dataset II contains 14 malware, each from a different family, which have been kindly provided to us by the authors of CYBERPROBE. Each malware corresponds to one of the 14 fingerprints evaluated in the CYBERPROBE paper [59]. We use Dataset II to compare the accuracy of the fingerprints produced by AUTOPROBE with the ones produced by CYBERPROBE.

6.4.2 Experiment Setup

For the malware execution phase we run the malware for 5 minutes each on a virtual machine with Intel Core Duo 1.50GHz CPU and 8 GB memory. Each run outputs an execution trace that serves as the starting point for the fingerprint generation components.

6.4.3 Experiment on Probe Generation

Set	**Type**	**Samples**	**R/O**	**# of Probes**	**Variable**	**Constant**	CYBERPROBE
I	R	14	35/42	22	13(59%)	9 (41%)	N/A
I	NR	10	152/152	11	9(81%)	2 (19%)	0
II	R	9	113/183	37	21(57%)	16(43%)	37(100%)
II	NR	10	121/121	15	8(54%)	7 (46%)	0

Table 6.1: Probe Generation Results

In Table 6.1, we summarize the results from the probe generation. We collect malware's execution/network traces and conduct the analysis. First, AUTOPROBE analyzes the network traces, extracts all the malware's network requests, and filter out those requests sent to domains in the Alexa top 10,000 list [1]. The number of remaining and original requests are shown in Table 6.1 in the column of R/O (means Remaining/Original requests). Then, for each dataset, we break the malware into two groups corresponding to whether at least one request received a response from a remote server (R:ResponseSeen), or all requests failed to receive a response (NR: NoResponse). For each group it shows the number of requests produced by the malware in the group during the executions and the number of probes produced by AUTOPROBE, split into probes that contain some variable parts and those that have

only constant parts. The last column shows the maximum number of probes that CYBERPROBE can produce for the group.

All requests are HTTP and on average it takes AUTOPROBE 14.3 minutes to analyze/process one execution trace, relatively slow but a reasonable cost for off-line analysis tools. AUTOPROBE generated a total of 85 fingerprints/probes for all 24 malware families in the two malware datasets or 3.5 fingerprints per malware family. Since multiple requests may be generated by the same execution path, the total number of probes is smaller than the number of requests captured on the network. We also observe that the majority of generated probes contain some variable parts. This means that dynamic binary analysis enables AUTOPROBE to extract more complete probe generation functions than network-based approaches, because the variable parts in the probe generation functions provide higher coverage.

Note that on both datasets, AUTOPROBE can generate fingerprints for all the malware, even those with no response, for which CYBERPROBE cannot. This demonstrates a clear advantage of AUTOPROBE. For the samples with a response in Dataset II, CYBERPROBE is able to generate a fingerprint similar to AUTOPROBE. However, for 57% of those, AUTOPROBE produces probes construction functions with variable fields rather than concrete probes in CYBERPROBE. Thus, AUTOPROBE probe construction functions are potentially more accurate. We also find 4 cases in which requests clustered together by CYBERPROBE are indeed generated by different logic in the malware. Thus, they should have been considered different as their responses are not guaranteed to have the same format.

6.4.4 Experiment on Classification

For classification function construction in both datasets, AUTOPROBE generates a total of 59 classification functions for all *ResponseSeen* cases and 26 for the *NoRe-*

sponse cases. For the *ResponseSeen* cases, the detection requires that all symbolic equations in the classification function match, so AUTOPROBE can finish matching when any of the equations fails to match. For the *NoResponse* cases, it calculates the suspicious score based on the matching results for all equations. For efficiency, our scanner records the response traffic and conducts a offline matching.

In Table 6.2, we show the classification function efficiency. We measure the time consumed for response matching/classification for handling a batch of 1,000 responses. For *ResponseSeen* cases, on average, a malware server classification function consists of 19 equations and takes 259 ms to fulfill the matching. The worst case is one malware server classification function that consists of 36 equation comparisons (CP) and it takes 757 million seconds to parse 1,000 responses. For *NoResponse* cases, a malware server classification function consists of more equations (52 on average) and takes 981 ms to fulfill the matching, 3x times more than the *ResponseSeen* cases. Overall, when classifying responses from Internet-wide scanning (Section 6.4.8), our classification component takes an average of 5 hours to analyze around 71 million responses.

Scheme	**WC(CP)**	**WC(ms)**	**BC (CP)**	**BC (ms)**	**AVG(CP)**	**AVG(ms)**
R	36	757	9	102	19	259
NR	67	1,923	37	483	52	981

Table 6.2: Efficiency of Classification Functions (time measured when handling 1000 continuous responses). CP: number of equation comparisons, WC: Worst Case, BC: Best Case

6.4.5 Case Studies

In this section, we study some probes generated by AUTOPROBE for real-world malware samples.

Bamital. Bamital is a well-known malware family for click-fraud attacks. To achieve the attack goal, Bamital communicates with its C&C server and visits multiple websites in a browser instance to simulate a real user visiting those websites. In our test, we captured one request as shown in Figure 6.9.

Through analyzing the requests, we find there are three variable parts in the request: (1) requested file name: `m.php` (2) `os` field which is obtained from the system call `GetVersionEx` (3) `host` field which is the output of a customized domain generation algorithm (DGA).

```
GET/[%1]?subid=61&pr=1&os=20&id=8BBFF356C9BA
905540BBB48D98C90697&ver=[%2] HTTP/1.0
Host: rigecejefuduseb.info
User-Agent: Mozilla/4.0 (compatible; MSIE
7.0; Windows NT 5.1)
Pragma: no-cache

[%1] = slice_0(random)
[%2] = slice_1(os_version)
[%3] = slice_2(time)
```

Figure 6.9: Probe for Batimal Trojan

In our test, AUTOPROBE did not receive the C&C server's response because it is no longer alive. However, through feeding a crafted `HTTP/1.1 200 OK` response, AUTOPROBE captures malware's logic such as searching of strings `<a>` and `<b>` in

the response. Based on the different response content, we also observe that malware constructs different requests to download new binary files.

Hence, our classification function first requires a successful connection with the `200` status code. Then AUTOPROBE continues on searching the string of `<a>[.*]</a>` and `<b>[.*]</b>`. If AUTOPROBE finds any string exists in the response, it will classify the remote server as a suspicious server.

Taidoor. Taidoor is a well-known malware family that has been consistently used in targeted attacks [78]. The C&C control logic of Taidoor also overlays on the HTTP protocol. It starts from one request shown in Figure 6.10.

AUTOPROBE first captures the file variable part, which is randomly generated and the length of the filename is limited to 5 characters. Meanwhile, the `id` field is built by the output of the library call `GetAdaptersInfo`, which could be the host's MAC address. When malware parses the response, the malware re-uses this `id` (its MAC) as the key to decode the response to receive the command from the botmaster. This `id` thus makes an interesting connection/correlation between the request and the response in the analysis. We consider such correlation between request and response could be an appropriate strong classification to detect the C&C server. Hence, our classification function conducts two-step operations: decode the data with the known key (the same from the request) and determine whether the decoded data is a valid ASCII string.

Sality. For Sality, we generate several requests of querying files `spm/s_tasks.php`, `logos_s.gif` and `231013_d.exe`. For the request of the `231013_d.exe` executable, the downloaded file will be directly executed. Thus, we takes the set of three file requests and three successful connections as the classification criteria. Any server hosting these three files in the same URL paths could be a possible Sality server.

```
Get /[%1].php?id =[%2] HTTP/1.1
User-Agent: Mozilla/4.0 (compatible: MSIE
6.0; Windows NT 5.1; SV1)
HOST: [%ip]
connection: keep-alive
Cache.Control: no-cache
[%1] = slice_0(5)
[%2] = slice_1(MAC_ADDRESS)
[%ip] = probe_ip
```

Figure 6.10: Probe for Taidoor Trojan

Other Malware. For Xpaj.B, AUTOPROBE generates one HTTP `POST` request with an encoded string, such as `POST /tRHmgD?kjBQMgpwJFLP=QOrbhqDjVeJmN`. The expected response from a valid malware server will start with a string `"filename="`. The remaining of the string will be used to create a file in the system. Our classification function is to search the string `"filename=[.*].bkr"` in the response payload. For Amonetize malware, we found one request which downloads a Fake VLC player from the server. Since AUTOPROBE finds the malware will verify the MD5 value of the downloaded file, we calculated the downloaded file's MD5 and makes the classification function for Amonetize. We observer similar activities from ZeroAccess which also download a windows executable from the remote server. In addition, we obtain another ZeroAccess's probe which queries a `links.php` file and visit all URLs in the response. This is likely that the C&C server distributes tasks of click-fraud activities. Hence, AUTOPROBE generated classification function will consider the server whose response upon specific request contains a list of URLs as a suspicious server.

6.4.6 Experiment Setup for Scanning

We conduct network scans using 5 machines of variable configurations. All machines run GNU/Linux Ubuntu 12.1 LTS with dual core 2.2 GHz CPUs and the memory configuration ranges from 2 GB to 16 GB. Our distributed scanning tasks are proportional to the number of scanners. On each scanner, we use multiple threads to further distribute scanning tasks. The performance of scanning is primarily limited by the speed of the network card in use and the CPU resource on the scanner. All the scanning experiments are conducted in the time period from November 4th, 2013 till November 11th, 2013.

6.4.7 Experiment on Localized Scanning

As we mentioned earlier, AUTOPROBE generated totally 85 probes for 24 malware families. To test the effectiveness of these probes, we perform 24 localized scanning first for each malware family.

Target Network Range. We first scan the network ranges that have been observed in the past to host some malicious servers. According to the provider locality property of malicious servers found in [59], these network ranges are more likely to find malicious servers than other regions on the Internet. We start with a seed set of 9,450 malware server IPs collected from MalwareDomainList.com as well as detected malicious servers provided by the authors of [59]. We then expand the IP list to include their network neighbors, i.e., those in the same /24 subnets and those from the BGP route information[†]. In this way, we have collected totally around 2.4M IPs for our localized scanning.

[†] We obtain the most specific BGP route that contains each seeds IP address.

ID	Set	Port	#	Time	Resp.	Found	Known	New	VT	MD	VQ
1	II	80	3	2.3h	64%	6	4	2	2	1	0
2	II	80	3	2.4h	64%	4	3	1	0	0	0
3	II	80	3	2.4h	64%	5	2	3	0	0	0
4	II	80	3	2.3h	64%	4	2	2	0	0	0
5	II	80	3	2.8h	64%	2	2	0	0	0	0
6	II	80	3	3.2h	64%	9	4	5	1	0	0
7	II	80	3	2.6h	63%	2	2	0	1	0	0
8	II	80	3	2.7h	63%	1	1	0	1	1	0
9	II	80	3	1.2h	63%	0	0	0	0	0	0
10	II	80	3	1.8h	63%	0	0	0	0	0	0
11	I	80	2	3.3h	64%	32	12	20	1	0	0
12	I	80	2	3.8h	64%	12	3	9	1	1	0
13	I	80	2	4.1h	64%	3	0	3	0	0	0
14	I	80	2	3.2h	64%	3	1	2	1	0	0
15	I	80	2	3.8h	64%	17	4	13	2	0	0
16	I	80	2	3.9h	64%	5	4	1	0	0	0
17	I	80	2	3.6h	64%	9	5	4	0	0	0
18	I	80	2	3.2h	64%	11	4	7	1	1	1
19	I	80	2	3.3h	64%	0	0	0	0	0	0
20	I	80	2	3.5h	64%	4	2	2	0	0	0
21	I	80	2	3.3h	64%	3	1	2	1	1	0
22	I	80	2	3.7h	64%	0	0	0	1	0	0
23	I	80	2	3.1h	64%	8	8	0	1	1	0
24	I	80	2	3.0h	64%	1	1	0	0	0	0
			TOTALS:			141	65	76	14	6	1

Table 6.3: Localized Scanning Results of AUTOPROBE. #: Number of Scanners

Result. In Table 6.3, we list the results of our 24 localized scanning tests. The left part of the table shows the scan configuration: the scan date, the malware dataset, the target port, the number of hosts scanned, and the number of scanners used (SC). The middle part of Table 6.5 shows the results: the scan duration, the response rate (Resp., i.e., the percentage of targets that replied to the probe), the number of total malicious servers found, the number of found malicious servers already in the seed set, and the number of new malicious servers (not in the seed set).

The results show that AUTOPROBE can efficiently finish scanning 2.4 million IPs with two parallel scanners in about 3 hours. Throughout the 24 scans, AUTOPROBE has identified a total of 141 malicious servers, among which 65 are known (in the seed set) and 76 are new (previously unknown) malicious servers.

We also compare our results with some existing malicious domain blacklists. In the experiment, we compare our results with three popular anti-malware blacklist services: VirusTotal [88] (VT), Malware Domain List [29](MD), URLQuery [87](UQ).

The best coverage is achieved by VirusTotal, which knows 9.9% of the servers found by AUTOPROBE (14/141). URL Query knows 6(4.25%) servers and Malware domain list knows only 1(0.01%) malicious servers. In this case, AUTOPROBE detects 8 times more malicious servers than the best of these blacklist services, clearly demonstrating that AUTOPROBE is an effective scheme for detecting malicious servers.

HID	**Type**	**Start Date**	**Port**	**Targets**
1	I	2013-11-04	80	2,528,563,104
# Scanners	**Rate(pps)**	**Time**	**Live Hosts**	
4	60,000	2.9h	71,068,585 (2.8%)	

Table 6.4: Horizontal Scanning Results

6.4.8 Experiment on Internet-wide Scanning

We next conduct another test of internet-wide scanning and compare the results with CYBERPROBE. To minimize the impact to the whole Internet because of our scanning while still clearly verifying the effectiveness of AUTOPROBE, instead of scanning all fingerprints, we select three malware families (soft196, ironsource, optinstaller) that have good results shown in CYBERPROBE [59] for the comparison

with AUTOPROBE in the new test.

Since these 3 malware families use HTTP C&C, we first perform an Internet-wide horizontal scan of hosts listening on the target port 80. This horizontal scan is summarized in Table 6.4. Before this horizontal scan, we collected the BGP table from RouteViews on November 3, 2013 and computed the total number of advertised IP addresses after removing overlaps. This yields to a total of around 2.5 billion hosts as the target of our horizontal scan.

On November 4, 2013 we performed the horizontal scan of those addresses on port 80/TCP using four scanner machines. We limit the scan rate to 60,000 packets per second (pps) for good citizenship. The scan takes 2.9 hours and finds 71 million live hosts listening on that port (80).

After obtaining this 71 million live HTTP server list, we performed 3 scanning using our AUTOPROBE and a copy of CYBERPROBE obtained from the authors (together with the fingerprints) for the three selected malware families. Table 6.5 summarizes the comparison. The top part of the table has the results for the CYBERPROBE scans and the bottom part the results for AUTOPROBE. Each row corresponds to one scan. The scan identifiers (CP-x for CYBERPROBE and AP-x for AUTOPROBE) imply different setup for this experiment. Similarly as in the localized scanning, we also compare our results with popular blacklist databases, VirusTotal (VT) [88], URLQuery (UQ) [87], and Malware Domain List (MD) [29] in the right part of Table.

The results show that for every malware family the fingerprints produced by AUTOPROBE find more servers than the one produced by CYBERPROBE. Overall, AUTOPROBE has found 54 malware servers, versus 40 malware servers found by CYBERPROBE, which represents a 35% improvement.

Finally, we also conduct three additional Internet-wide scans for probes that

ID	Port	Fingerprint	#	Time	Resp.	Found	Known	New	VT	MD	VQ
CP-1	80	soft196	2	24.6h	91%	9	8	1	1	0	0
CP-2	80	ironsource	2	24.6h	92%	11	7	4	4	1	0
CP-3	80	optinstaller	2	24.6h	90%	20	4	16	6	0	0
			Cyberprobe **TOTALS:**			40	19	21	11	1	0
AP-1	80	soft196	2	2h	90%	13	8	1	3	1	0
AP-2	80	ironsource	2	24.6h	91%	17	6	4	9	2	0
AP-3	80	optinstaller	2	24.6h	92%	24	5	16	9	2	0
			Autoprobe **TOTALS:**			54	19	21	21	5	0

Table 6.5: Result of Internet-side Scanning. Here CP-x denotes Cyberprobe and AP-x denotes Autoprobe.

cannot be generated by Cyberprobe, i.e., those from the `NoResponse` malware server cases. The result is summarized in Table 6.6. As we can see, Autoprobe can detect 48 malware servers, with most of them (83%) are new servers. Compared with Cyberprobe, which cannot generate any probe for *NoResponse* cases, Autoprobe clearly has unique advantage and thus complements existing work very well.

ID	Port	Fingerprint	#	Time	Resp.	Found	Known	New	VT	VQ	MD
AP-1	80	Sality	5	12.1h	90%	23	3	20	1	0	0
AP-2	80	Taidoor	5	13.2h	91%	14	4	10	2	1	0
AP-3	80	Bamital	5	12.6h	92%	11	1	10	2	0	0
		Autoprobe **TOTALS:**			48	8	40	3	1	0	0

Table 6.6: Additional 3 Scanning Results of Autoprobe for `NoResponse` Cases

False positives and False Negatives. Given the lack of perfect ground truth, to measure the false positives of our detection we check whether the server can successfully trigger client-side malware's malicious logic and establish successful communication with the remote server. Hence, for each detected server, we conduct another round of verification by redirecting malware's request to the detected servers and monitor malware's execution afterwards. If the malware's execution goes into the

behaviors we found in the analysis phase, we think it is true positive case. In our test, we do not find any false positive case in our test. To measure false negatives, we use the detection result of CYBERPROBE as the ground truth. The result shows that AUTOPROBE can correctly detect all the results in CYBERPROBE using different signatures for the same families. We further discuss potential false positives and false negatives in Section 6.5.

6.5 Limitation

In this section, we discuss some limitations and possible evasions of AUTOPROBE.

6.5.1 Responses Check

Our classification function construction assumes that the malware will behave differently when receiving valid and invalid responses from remote servers. If the malware violates this assumption, i.e., performs no checks or only cursory checks on the responses, the generated fingerprints may produce false positives when probing benign servers. However, this situation does not arise in our examples and we believe it is unlikely as it would be extremely easy to infiltrate such C&C protocol.

6.5.2 Classification Function through Code Reuse

The classification function produced by AUTOPROBE is a logic expression applied on the response or the output of a parser on the response. Those expressions are difficult to extract if the variables follow non-linear relations. In those cases we could apply binary code reuse techniques [13, 49] to directly (re)use the malware's reponse handling code. In the extreme case, AUTOPROBE could rerun the malware in the controlled environment on the responses received from target servers. Obviously, such approaches are expensive, so they are better used only when our current approach cannot determine a symbolic expression.

6.5.3 Fuzzing

The fingerprints produced by AUTOPROBE use valid probes that satisfy the C&C protocol grammar because the probe construction functions that generate them have been extracted from the malware's request generation logic. However, for some families it may be possible to generate additional fingerprints using invalid probes that do not satisfy the C&C grammar but still trigger a distinctive response from the C&C servers. Invalid probes are easier to be identified by the C&C server managers but may be useful when the C&C masks as a benign protocol. When a live C&C server is known, AUTOPROBE could be enhanced with a fuzzing approach that uses the semantic information extracted during probe generation to modify valid probes into invalid and test them against the C&C server.

6.5.4 Possible False Positive and False Negative

As discussed in Section 6.4, we do not find any false positive and false negative cases in our detection result. We think it is because we apply very strict criteria to determine whether it is a malicious server or not. For example, we ensure the response can indeed trigger malware to download malicious file or send some response. However, since our criteria of detecting malicious server purely depends on malware's behaviors, lacking of full and precise understanding of malware logic may mislead our detection. For example, malware may download one malicious file from the server and its continual logic may depends on the success of downloading. However, if our analysis tool cannot capture malware's behavior using such file in the limited monitoring time, AUTOPROBE may directly treat any server hosting this file as the malicious one. We think the root cause of such false positive/negative is because the limitation of dynamic analysis: we can only observe partial result of malware logic. To improve and provide more accurate result, we should provide more analysis time

and more code coverage measurement in the real world deployment.

6.6 Related Works

Scanning the internet is one way to find large-scale network-level vulnerabilities. Provos et al. scanned Internet to identify vulnerable SSH servers through vulnerability signatures [64]. Dagon et al. [27] scanned DNS servers on Internet to find those providing incorrect resolutions. Heninger et al. [43] scanned the Internet to find network devices with weak cryptographic keys. All these studies apply some widely-known signatures to achieve the purpose.

Different from them, active probing to detect network-based malware has been proposed and discussed in several previous work [59, 62, 98, 40, 4]. In [40], Gu et al. proposed to actively send probing packets through IRC channels. Zmap [30] is another internet-wide scanner which is efficient enough to scan the whole internet in less than 45 minutes. However, it targets to test the aliveness of remote hosts instead of detecting possible malicious servers.

PeerPress [98] is one related work that also adopts dynamic malware analysis to find P2P malware's informed active probs. Nevertheless, as we have stated the difference earlier, the target of such probing is on the malware samples that actively open the port for communication, such as P2P malware and Trojan Horse. AUTOPROBE targets at remote malicious servers and we assume the server-side logic is not available for analysis in collected binaries, a different assumption from PeerPress.

Fingerprinting network applications is a widely studied topic. Botzilla [69] is a method for detecting malware communication through repetitively recording network traffic of malware in a controlled environment and generating network signatures from invariant content patterns. AUTOPROBE has a different goal of fingerprinting malicious servers and adopts binary-level analysis to find the invariant part in

packets.

FiG [16] proposed a framework for automatic fingerprint generation that produces OS and DNS fingerprints from network traffic. In contrast, AUTOPROBE applies a different approach for automatic fingerprint generation that takes as input a malware sample and applies dynamic binary analysis on the malware's execution.

There are multiple existing studies that discuss effective and efficient techniques for malware analysis. Such techniques include taint analysis [48], enforced execution [93], path exploration [57], program slicing [13], symbolic execution [89] and trace alignment [46]. AUTOPROBE applies many of these techniques in our new problem domain in a novel way to automatically generate network fingerprints.

Among all studies on binary analysis, protocol reverse engineering work, such as [14, 25, 17], is also closely related to AUTOPROBE. We adopt similar approach as in [14] to figure out the semantics meanings of malware's request. However, one difference between AUTOPROBE and existing work is that AUTOPROBE does not attempt to understand the complete protocol of malware's communication, and AUTOPROBE uses many other different techniques to aid the generation of fingerprints.

6.7 Summary

In this chapter, we present AUTOPROBE as an automatic framework to generate active probing fingerprints for Internet-wide malicious server detection. Our approach employs dynamic malware analysis to improve the effectiveness and efficiency of existing work. In our extensive Internet-scale scanning, AUTOPROBE outperforms the existing state-of-the-art system in discovering more malicious servers.

7. LESSONS LEARNED FROM NEW MALWARE ATTACKS

In this dissertation, we have introduced four systems: GoldenEye, AutoVac, PeerPress and Autoprobe. In this chapter, we try to answer two questions about our systems: How are these systems related to and different from each other? What lessons have we learned from designing these systems to conduct malwware analysis?

7.1 Summary of Our Malware Analysis System

In Table 7.1, we briefly summarize the features of our malware analysis systems. We study them through their design goal, malware coverage, techniques such as whether they apply tainted analysis, symbolic execution and branch evaluation or not, and performance such as effectiveness and efficiency. Next, we present each part in details.

	GoldenEye	AutoVac	PeerPress	Autoprobe
Goal	Analysis	Host Protection	P2P Malware Detection	Malicious Servers
Guideline	Large-Scale	Large and In-depth	In-depth	In-depth
Coverage	General	General	Specific	Specific
API Hooking?	Yes	Yes	Yes	Yes
Tainted Tracking?	Yes	No	Yes	Yes
Symbolic Execution?	No	No	Yes	Yes
Branch Prediction/Evaluation?	Yes	No	Yes	Yes
Effectiveness	Possible False Positive/Negative	Possible False Negative	Possible False Negative	Possible False Positive/Negative
Overhead	Low	Low	High	High

Table 7.1: Summary of Our Malware Analysis Systems

7.1.1 Design Goal, Guideline and Coverage

Our four systems have different design goals. Even though each system applies dynamic malware analysis, the problems and challenges are different. For GoldenEye, it is designed to improve the effectiveness of existing malware analysis system. Specifically, it is a system for large-scale malware analysis. That is why we focus on improving its efficiency in design. AutoVac is one system designed to generate a vaccine for host protection. Therefore, it adapts both large-scale and in-depth analysis at the same time. In large-scale analysis step, we need to quickly filter out those samples which possibly have vaccines. In in-depth analysis, we extract vaccines from malware binary. Therefore, we design different techniques for two phases. For PeerPress and Autoprobe, these two systems follow the guideline of in-depth analysis with large-scale detection, which means we conduct delicate analysis on malware's communication logic and, after we extract the probings, our detection is deployed on internet-wide detection. The difference between PeerPress and Autoprobe is their targets, i.e.,, PeerPress is for P2P malware and Autoprobe is for malicious servers. As a result, their coverage is limit to specific category of malware families.

7.1.2 Technique Design

Different design goals affect the techniques each system applies. In previous works, three representative malware analysis techniques, tainted analysis [71], symbolic execution[71] and branch prediction or evaluation [98] have been proposed for different purposes. Tainted analysis is one data-flow tracking technique and it can analyze program's data processing logic. However, it normally consumes higher overhead. Similar situation applies to symbolic execution, which applies analysis on all control and data flow. To the contrary, the branch prediction and evaluation, which we proposed in our previous paper [98], is an alternative solution which exchanges

accuracy with lower overhead.

For our systems, GoldenEye applies branch evaluation and modified version of tainted analysis to ensure its efficiency. AutoVac only uses API hooking which makes it the most efficient system among the four systems. As the goal of both PeerPress and Autoprobe is to deeply understand malware, they apply all three techniques to refine the result.

7.1.3 Performance

All these four system have false negative cases. It is because they cannot find all possible malware paths by using dynamic analysis. It is an intrinsic limitation of all dynamic malware analysis schemes. For false positives and overhead, four systems vary in different cases.

The GoldenEye may generate some false positive cases because of the inaccuracy of branch prediction. However, the distributed version of GoldenEye, which consumes more computing resources, has a better trade-off. AutoVac can produce result without false positives, however, its false negative rate may be higher because our vaccine selection criteria is extremely stringent to ensure that the vaccine will not affect the normal use of benign software. For PeerPress and Autoprobe, they consume higher analysis overhead but with better effectiveness. However, Autoprobe may also generate some false positive cases because we only use the client-side logic to deduce server-side logic since the server-side program is not available for analysis. It unavoidably brings inaccuracy in the result.

7.2 Lessons Learned

As we have highlighted the challenges in each chapter, the arm races between malware authors and malware analysts may never end. It stimulates we improve our analysis schemes further for all new attacks. From designing and implementing all

four systems, we have learned the following important lessons.

7.2.1 Applying Delicate Analysis in Large-scale Malware Analysis is Feasible

Tainted analysis and symbolic execution are *expensive* analysis techniques and analysts may thought they are not applicable in large-scale analysis. However, our practice in GoldenEye proves applying these techniques on large-scale malware analysis is also feasible in real-world. It first owes to the improvement of computing infrastructure and new computing models. For example, our distributed computing model in GoldenEye greatly improves the efficiency of the original design. Secondly, we also need to do some modification of these techniques to let them adept to the new scenario. As we discussed, GoldenEye uses a modified tainted analysis and simplifies the original procedures. The main design modification is to prevent analysis tasks from recursively happening. Hence, we design an efficient scheme in GoldenEye to stop the analysis at some branch point, which limits its recursive overhead. We consider it is a good design example to combine delicate analysis with large-scale analysis.

7.2.2 Revisiting of Vaccination Idea is Worthwhile

Our motivation of AutoVac is originated from David Ferbrache's 1992 book [34]. Unfortunately, he only briefly talked about high level features of vaccination and did not systematically explore this problem further. Revisiting the vaccination gives us a lot of research inspiration. Previous limitation of the work, i.e.,malware choose not to share some common feature, changes in current situation. As we mentioned, the internet-propagated malware chooses to use same infection marker to prevent duplicate infection, which violates the previous work's assumption.

It is a good lesson for us to revisit some unsolved problems in previous works and think them in the new contexts. We want to share this lesson with other researchers

to stimulate more scientific thinking of previous works.

7.2.3 Combing Network-based Detection with Host-based Analysis is Promising

The main design of PeerPress and Autoprobe is to combine the network-based detection with host-based analysis. The main merit of network-based is to save great resource for detection. In this case, we do not need to install AV-tools, update ever-increasing signature database and perform real-time monitoring. However, the limitation of network-based detection is the quality of signatures. The host-based malware analysis can perfectly fill the gap. It actively analyzes the malware's network communication logic which is different from existing schemes which passively monitors the malware's network traffic. In all, our proposed informed active probing, which combines network-based detection and host-based analysis, is more robust, more lightweight and large-scale deploy-able compared with tradition network-based and host-based detection approaches.

8. CONCLUSION AND FUTURE WORK

8.1 Conclusion

After discussion of all our work and lessons we learned, we summarize the dissertation in this chapter. In this dissertation, we discuss three kinds of new evolutions in malware intrusion.

In Chapter 3, we discuss the *target evolution* that malware chooses to launch attacks targeting at a set of specific hosts. The greatest challenge it brings is *how can defenders effectively and efficiently analyze such malware instances* given a large number of samples collected everyday. Our proposal is a novel analysis system, GoldenEye, to analyze targeted malware. From the discussion of targeted malware's characteristic, we develop an automatic approach to extract malware targeted environments before the traditional dynamic analysis. We propose some dynamic analysis techniques, such as parallel environment spaces construction, speculative execution in parallel spaces and branch evaluation to solve the technique challenges of the problem. Then we analyze the real world malware using GoldenEye, and we show our schemes can work on real-world malware corpus and achieve a better performance trade-off compared with the existing works. More importantly, our initial work may stimulates more following research to discuss more systematic method for analyzing targeted malware threat.

In Chapter 4, we present AutoVac, a new complementary malware defense system that aims to extract possible malware vaccines from given malware samples. The malware vaccine is an effective protection scheme which uses the strength of malware's *technique evolution* against itself. Malware vaccine can be used to build an immune system at an end host to defend against the specific malware's infection.

To demonstrate the real-world practicability, we conduct experiments on a large set of real-world malware samples and successfully extract working vaccines for many families. Our result is very encouraging as a proof-of-concept study of malware vaccine generation. We believe it is an appealing complementary technique in defending massive malware intrusion.

The last topic we covered in this dissertation is how to handle the new challenges brought by malware's *communication evolution*.

In the first step, we focus on malware which provides server-side functionality, such as P2P malware. In Chapter 5, we discuss a novel, two-phase detection framework that seamlessly bridges host-level dynamic binary analysis and network-level informed active probing techniques. It can detect P2P malware and beyond, as long as the malware has malware control birthmarks. We developed new techniques such as ICE to tackle our research challenges, and we implemented a prototype system, PeerPress, to demonstrate the real-world utility. Our results on real-world dataset are very encouraging. PeerPress demonstrates an important step toward *proactive* malware detection and defense (instead of passive monitoring), a direction worth more attention from the security research community.

In the second step, we concentrate on internet malware while only one-side of malware, client-side, binary is available for analysis. The problem is challenging because we target to detect the malicious server only based on partial network communication. To solve the problem, In Chapter 6, we present Autoprobe as an automatic framework to detect Internet-wide malicious servers. Our approach employs dynamic malware analysis to improve the effectiveness and efficiency of existing work. The analysis can help expose more requests, identify the fingerprint response, and assist in efficient detection. Furthermore, Autoprobe proposes new solutions for some real-world challenges such as none-alive servers. We also show that Autoprobe

can generate more accurate network fingerprints for malicious servers probing. In our extensive Internet-scale scanning, AUTOPROBE outperforms the existing state-of-the-art system in discovering more malicious servers.

8.2 Future Work

The arm races between defenders and attackers will never end. Naturally, because of the nature limitation of dynamic analysis, our proposed schemes is definitely not perfect. As discussed in previous limitation sections, all our analysis could be impaired because of implicit data flow, unsolvable conditions and strong flow distortion and obfuscation.

Fortunately, our dynamically analysis framework is designed to be highly extendible. For future work, we can incorporate more analysis components to achieve more effective and efficient analysis.

One possible extension is integrating a *localized* static analysis component. In this way, we can use the advantage of dynamic analysis to evade obfuscation and apply static analysis to obtain a complete view of malware behaviors.

The second components is encryption cracking component. Nowadays, malware widely use encrypted communication. To treat the encryption handling as independent analysis step can efficiently mitigate the complexity of our dynamic analysis. Some existing works, such as[15] has proposed some useful ideas of implementing such components.

Lastly, a complete protocol reverse engineering component can provide a great help for some complex and advanced malware families. For example, PRE component can improve PEERPRESS to handle more complex multiple rounds of validation and long stateful protocol interactions.

REFERENCES

[1] Alexa. Top Domains. http://www.alexa.com/, 2005.

[2] Amonetize Adware. Amonetize. http://greatis.com/cleanvirus/remove-malware/a-variant-of-win32amonetize-q.htm, 2010.

[3] Anubis. Anubis: Analyzing unknown binaries. http://anubis.iseclab.org/, 2007.

[4] Ofir Arkin. A remote active os fingerprinting tool using icmp. *;login: The USENIX Magazine*, 27(2), November 2008.

[5] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient detection of split personalities in malware. In *Proceedings of 17th Annual Network and Distributed System Security Symposium (NDSS 2010)*, San Diego, CA, Feburary 2010.

[6] Bamital Malware. https://now-static.norton.com/now/en/pu/images/Promotions/2013/Bamital/bamital.html, 2013.

[7] Leyla Bilge and Tudor Dumitras. Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, October 2012.

[8] Phabot Botnet. Phabot. http://www.secureworks.com/research/threats/phatbot/?threat=phatbot, 2008.

[9] Qakbot Botnet. Qakbot. http://www.symantec.com/connect/blogs/w32qakbot-under-surface, 2012.

[10] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations

with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Boston, MA, August 2007.

[11] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Analysis and Defense*, volume 36, pages 65–88. Springer, Neu-Isenburg, Germany, 2008.

[12] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the Conference on Computer Aided Verification*, Snowbird, Utah, July 2011.

[13] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary Code Extraction and Interface Identification for Security Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2010.

[14] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of ACM Conference on Computer and Communications Security*, Chicago, IL, November 2009.

[15] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babi ć, and Dawn Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 413–425, Chicago, Illinois, USA, November 2010. ACM.

[16] Juan Caballero, Shobha Venkataraman, Pongsin Poosankam, Min Gyung Kang, Dawn Song, and Avrim Blum. Fig: Automatic fingerprint generation.

In *Proceedings of Network and Distributed System Security Symposium*, San Diego, CA, February 2007.

[17] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007.

[18] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of International Conference on Dependable Systems and Networks*, Anchorage, AK, June 2008.

[19] Paolo Milani Comparetti, Guido Salvaneschi, Clemens Kolbitsch, Christopher Kruegel, Engin Kirda, and Stefano Zanero. Identifying dormant functionality in malware programs. In *Proceedings of 31st IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2010.

[20] Symantec Corp. Symantec Internet Security Threat Report. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`, 2010.

[21] Symantec Corp. The Nitro Attacks: Stealing Secrets from the Chemical Industry. `http://www.symantec.com/security_response/whitepapers.jsp`, 2012.

[22] Symantec Corp. Symantec Intelligence Security Report. `http://www.symantec.com/security_response/publications/threatreport.jsp`, 2013.

[23] Trend Micro Corp. Trends in targeted attacks. `http://www.trendmicro.com/cloud-content/us`, 2012.

[24] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 133–147, Brighton, United Kingdom, October 2005.

[25] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, Boston, MA, August 2007.

[26] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. tupni: Automatic reverse engineering of input formats. In *Proceedings of ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2008.

[27] David Dagon, Chris Lee, Wenke Lee, and Niels Provos. Corrupted dns resolution paths: The rise of a malicious resolution authority. In *Proceedings of Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[28] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)*, pages 51–62, Alexandria, Virginia, USA, November 2008.

[29] Malicious Domains. Malware domain list. `http://malwaredomainlist.com/`, 2013.

[30] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proceedings of 22nd Usenix Security Symposium*, Washington, D.C., August 2013.

[31] DynamoRIO. DynamoRIO Binary Instrumentation. http://dynamorio.org/, 2008.

[32] Bellard Fabrice. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Berkeley, CA, USA, Feburary 2005.

[33] Nicolas Falliere. Sality: Story of a peer-to-peer viral network. Technical report, 2011.

[34] David Ferbrache. *A pathology of computer viruses*, pages 43–47. Springer-Verlag, Berlin, Germany, 1992.

[35] Spam Fighter. Cybercriminals Making Sality Virus More Complex. http://www.spamfighter.com/Cybercriminals-Making\ \-Sality-Virus-More-Complex-16068-News.htm, 2011.

[36] Alexander Gostev. 2010: The year of the vulnerability . http://www.net-security.org/article.php?id=1543, 2010.

[37] Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent ByungHoon Kang, and David Dagon. Peer-to-peer botnets: overview and case study. In *Proceedings of the first Conference on First Workshop on Hot Topics in Understanding Botnets (HotBot'07)*, Cambridge, MA, August 2007.

[38] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the 17th USENIX Security Symposium (Security'08)*, San Jose, CA, August 2008.

[39] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. BotHunter: Detecting malware infection through ids-driven dialog correlation.

In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, Boston, MA, August 2007.

[40] Guofei Gu, Vinod Yegneswaran, Phillip Porras, Jennifer Stoll, and Wenke Lee. Active botnet probing to identify obscure command and control channels. In *Proceedings of 2009 Annual Computer Security Applications Conference (ACSAC'09)*, Honolulu, Hawaii, December 2009.

[41] Guofei Gu, Junjie Zhang, and Wenke Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[42] Duc T. Ha, Guanhua Yan, Stephan Eidenbenz, and Hung Q. Ngo. On the effectiveness of structural detection and defense against p2p-based botnets. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Lisborn, Portugal, April 2009.

[43] Nadia Heninger, Zagir Durumeric, Eric Wustrow, and J.Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, August 2012.

[44] Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, 2012.

[45] Márk Jelasity and Vilmos Bilicki. Towards automated detection of peer-to-peer botnets: On the limits of local approaches. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, pages 3–3, Berkeley, CA, August 2009.

[46] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of* 32nd *IEEE Symposium on Security and Privacy*, pages 347 –362, Oakland, CA, May 2011.

[47] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, August 2006.

[48] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of 18th USENIX Security Symposium*, Montréal, Canada, August 2009.

[49] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 29–44, Washington, DC, May 2010.

[50] Koobface Adware. Koobface. `http://www.symantec.com/security_response/writeup.jsp?docid=2008-080315-0217-99&tabid=2`, 2008.

[51] Bogdan Korel and Janusz W Laski. Dynamic program slicing. *Information Processing Letter*, 29(3):155–163, 1988.

[52] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: Using system-centric models for malware protection. In *Proceedings of the 17th ACM Conference on Computer and Commu-*

nications Security, CCS '10, pages 399–412, New York, NY, USA, November 2010.

[53] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 338–357, Berlin, Heidelberg, September 2011.

[54] Andrea De Lucia. Program slicing: Methods and applications. In *Proceedings of 1st IEEE International Workshop on Source Code Analysis and Manipulation*, San Jose, CA, May 2001.

[55] Bifrost Malware. Bifrost. `http://en.wikipedia.org/wiki/Bifrost_(Trojan_horse)`, 2006.

[56] Mebroot. `http://www.symantec.com/security_response/writeup.jsp?docid=2008-010718-3448-99`, 2011.

[57] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Washington, DC, 2007.

[58] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: Finding p2p bots with structured graph analysis. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 7–7, Berkeley, CA, USA, August 2010.

[59] Antonio Nappa, Zhaoyan Xu, M. Zubair Rafique, Juan Caballero, and Guofei Gu. Cyberprobe: Towards internet-scale active detection of malicious servers. In *Proceedings of Network and Distributed System Security Symposium*, San Diego, CA, Feburary 2014.

[60] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of of Network and Distributed System Security Symposium*, San Diego, CA, February 2005.

[61] Offensive Computing. Online malware database. `http://www.offensivecomputing.net/`, 2010.

[62] Jitendra Padhye and Sally Floyd. Identifying the tcp behavior of web servers. In *Proceedings of Annual Conference of the Special Interest Group on Data Communication*, San Diego, CA, August 2001.

[63] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. `http://mtc.sri.com/Conficker/`, 2009.

[64] Niels Provos and Peter Honeyman. Scanssh - scanning the internet for ssh servers. In *Technical Report CITI TR 01-13, University of Michigan*, Ann Arbor, MI, October 2001.

[65] Ramnit Malware. Ramnit. `http://en.wikipedia.org/wiki/Ramnit`, 2011.

[66] Dark Reading. Targeted Attack for 2012. `http://www.darkreading.com/views/2012-us-election-and-targeted-attack-pre/232900698`, 2012.

[67] Mike Reiter and Ting fang Yen. Traffic aggregation for malware detection. In *Procceedings of Detection of Intrusions and Malware and Vulnerability Accessment*, Paris, France, July 2008.

[68] Microsoft Research. Z3 EMT Solver . `http://research.microsoft.com/en-us/um/redmond/projects/z3/`, 2012.

[69] Konrad Rieck, Guido Schwenk, Tobias Limmer, Thorsten Holz, and Pavel Laskov. Botzilla: Detecting the phoning home of malicious software. In *Proceedings of 25th ACM Symposium on Applied Computing*, Sierre, Switzerland, December 2010.

[70] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of 22nd Annual Computer Security Applications Conference*, Miami Beach, FL, November 2006.

[71] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[72] Chengyu Song, Paul Royal, and Wenke Lee. Impeding automated malware analysis with environment-sensitive malware. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Security*, Berkeley, CA, Feburary 2012.

[73] Joe Stewart. Inside the Storm. `http://www.blackhat.com/presentations/bh-usa-08/Stewart/BH_US_08_Stewart_Protocols_of_the_Storm.pdf`, 2005.

[74] Elizabeth Stinson and John C. Mitchell. Towards systematic evaluation of the evadability of bot/botnet detection methods. In *Proceedings of of 2nd USENIX Workshop on Offensive Technologies*, San Jose, CA, July 2008.

[75] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich. Analysis of the storm and nugache trojans: P2P is here. In *;login Magazine*, New York, NY, August 2007.

[76] Symantec. Triage analysis of targeted attacks. http://www.symantec.com/threatreport/topic.jsp?id=malicious_code_trend, 2013.

[77] Peter Szor. *The Art of Computer Virus Research and Defense*, pages 240–245. Addison Wesley Professional, Boston, MA, 2005.

[78] Taidoor Malware. Xpaj.b malware. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp_the_taidoor_campaign.pdf, 2012.

[79] Ionut Trestian, Supranamaya Ranjan, Aleksandar Kuzmanovic, and Antonio Nucci. Unconstrained Endpoint Profiling (Googling the Internet). In *Proceedings of ACM SIGCOMM*, Hong Kong, China, August 2008.

[80] Duqu Trojan. Duqu. http://en.wikipedia.org/wiki/Duqu, 2012.

[81] IBank Trojan. IBank. http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Troj~IBank-B/detailed-analysis.aspx, 2011.

[82] MalC0de Trojan. malc0de. http://malc0de.com/database/, 2012.

[83] Neloweg Trojan. Neloweg. http://www.symantec.com/security_response/writeup.jsp?docid=2012-020609-4221-99, 2013.

[84] NuclearRAT Trojan. NuclearRAT. http://en.wikipedia.org/wiki/Nuclear_RAT, 2010.

[85] Sality Trojan. Sality. http://www.symantec.com/security_response/writeup.jsp?docid=2006-011714-3948-99, 2006.

[86] Zeus Trojan. Zeus. http://en.wikipedia.org/wiki/Zeus_(Trojan_horse), 2008.

[87] Urlquery. http://urlquery.net/, 2010.

[88] Virustotal. Virustotal online malware classification. `http://www.virustotal.com/`, 2013.

[89] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *In Procceedings of the 31st IEEE Symposium on Security and Privacy (Oakland'10)*, May 2010.

[90] Xiaofeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS)*, pages 37–46, Alexandria, Virginia, USA, November 2006.

[91] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proceedings of 16th ACM Conference on Computer and Communication Security*, Chicago, IL, USA, November 2009.

[92] Andre Wichmann and Elmar Gerhards-Padilla. Using infection markers as a vaccine against malware attacks. In *Proceedings of the 2nd workshop on Security of Systems and Software resiLiency*, Besancon, France, March 2012.

[93] Jerey Wilhelm and Tzi cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Procceding of International Symposium on Research in Attacks, Intrusions and Defenses*, Queensland, Australia, September 2007.

[94] Conficker Worm. Conficker C Analysis Report . `http://mtc.sri.com/Conficker/`, 2008.

[95] Stuxnet Worm. Stuxnet. `http://en.wikipedia.org/wiki/Stuxnet`, 2011.

[96] James Wyke. The zeroaccess botnet: Mining and fraud for massive financial gain. `http://www.sophos.com/en-us/why-sophos/our-people/technical-papers/zeroaccess-botnet.aspx`, September 2012.

[97] Xpaj.B Malware. Xpaj.b malware. `http://www.symantec.com/connect/blogs/w32xpajb-file-infector-vengeance`, 2011.

[98] Zhaoyan Xu, Lingfen Chen, Guofei Gu, and Christopher Kruegel. Peerpress: Utilizing enemies' p2p strength against them. In *Proceedings of 18th ACM Conference on Computer and Communications Security*, Raleigh, NC, October 2012.

[99] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. AUTOVAC: Towards automatically extracting system resource constraints and generating vaccines for malware immunization. In *Proc. of the 33rd International Conference on Distributed Computing Systems (ICDCS'13)*, Philadephla, PA, July 2013.

[100] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, CA, January 2010.

[101] Heng Yin, Dawn Song, Egele Manuel, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007.

[102] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'02/FSE-10)*, pages 1–10, Charleston, South Carolina, USA, November 2002.