A STUDY OF SELECTED SECURITY MEASURES AGAINST NON-LEGITIMATE USE OF CODE

A Dissertation in
Computer Science and Engineering
by
Yoon-Chan Jhi

© 2010 Yoon-Chan Jhi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2010
The dissertation of Yoon-Chan Jhi was reviewed and approved* by the following:

Peng Liu  
Professor of Information Sciences and Technology  
Dissertation Advisor, Co-Chair of Committee

George Kesidis  
Professor of Computer Science and Engineering  
Co-Chair of Committee

Sencun Zhu  
Assistant Professor of Computer Science and Engineering

Zan Huang  
Assistant Professor of Supply Chain and Information Systems

Raj Acharya  
Professor of Computer Science and Engineering  
Department Head

*Signatures are on file in the Graduate School.
Abstract

As many aspects of social and economic activities rapidly move to networks, computer security has never been more important. Many serious security issues have their origin in non-legitimate use of code. For example, code exploiting a vulnerability can be used in computer worms and viruses; code used without acquiring proper permission could conduct intellectual property theft; or code inserted with a malicious intention may be abused as spyware, botnet, or backdoors to cause many additional security problems.

In this study, we consider two types of security issues that can be caused by a non-legitimate use of program code. First, we investigate self-propagating code, computer worms. To defend against worm spread, we propose PWC, a proactive worm containment solution for enterprises. PWC can stop - instead of just slow down - an infected host from releasing worm scans after merely 4 scans. Motivated by the observation that a worm uses a sustained outgoing packet rate, PWC gains infection awareness seconds before a signature or filter can be generated. To overcome denial-of-service possibly caused by such characteristic indicators of infection, PWC develops two new white-host detection (uninfected host detection) techniques: (a) the vulnerability time window lemma, and (b) the relaxation analysis. PWC does not rely on content-based signatures and thus it can quickly contain polymorphic worms. PWC is also resilient to containment evasion. PWC is not sensitive to worm scan rate, and is not protocol specific. Due to white-host detection, PWC causes minimal denial-of-service. Evaluation based on real traces and worm simulations demonstrates that PWC significantly outperforms Virus Throttle [1] in terms of number of released worm scans, number of hosts infected by local scans, and denial-of-service effects.

Second, we study detection of illegally reused code. Reusing code can be done with malicious motives. Code theft is accomplished by cracking, reverse engineering, breaking into an enterprise network, spying, bribing, etc. In a sense, absence of effective methods to detect fraudulent use of program code has a negative influence to security. For a practical and effective method to detect software plagiarism,
we propose value based plagiarism detection system (VaPD). Based on an observation that some critical runtime values are hard to be replaced or eliminated by semantics-preserving transformation techniques, we introduce a novel approach to dynamic characterization of executable programs. Leveraging such invariant values, our technique is resilient to various control and data obfuscation techniques. We show how the values can be extracted and refined to expose the critical values and how we can apply this runtime property to help solve problems in software plagiarism detection. We have implemented a prototype with a dynamic taint analyzer atop a generic processor emulator, QEMU 0.9.1. Our experimental results show that our value-based plagiarism detection method successfully discriminates 34 plagiarisms obfuscated by SandMark, plagiarisms heavily obfuscated by Klass-Master, C programs obfuscated by Thicket, and executable files obfuscated by Loco/Diablo.

In summary, this study presents two security measures against non-legitimate use of program code. First, a worm containment technique is proposed to stop the spread of self-propagating malicious code within enterprise networks. Second, a software plagiarism detection technique is proposed as a practical measure to deter theft accompanied by software plagiarism activity. These two techniques contribute to uninterrupted communications as well as promote a more healthy and trustworthy environment for the software industry.
# Table of Contents

List of Figures ix  
List of Tables xi  
Acknowledgments xii  

## Chapter 1  Introduction 1  
1.1 Problems in This Study ................................................. 3  
1.1.1 Defense against Propagation of Computer Worms .............. 3  
1.1.2 Defense against Code Plagiarism .............................. 4  
1.2 Contributions of This Study ........................................... 4  
1.2.1 Worm Containment .................................................. 4  
1.2.2 Software Plagiarism Detection .................................... 5  
1.3 Outline .................................................................. 6  

## Chapter 2  Related Work 8  
2.1 Worm Containment ....................................................... 8  
2.1.1 Class A: Rate limiting ............................................. 8  
2.1.2 Class B: Signature-based worm scan filtering ............... 9  
2.1.3 Class C: Filter-based worm containment ..................... 9  
2.1.4 Class D: Payload-classification based worm containment ... 10  
2.1.5 Class E: Threshold Random Walk (TRW) scan detection ... 10  
2.1.6 Other related work ............................................... 10  
2.2 Software Plagiarism Detection ....................................... 11  
2.3 State of the Art ....................................................... 11  

## Chapter 3  PWC: A Proactive Worm Containment Solution for Enterprise Networks 14  
3.1 Introduction ............................................................. 14  
3.2 PWC Overview .......................................................... 17
3.2.1 Definition and Scope ........................................ 17
3.2.2 Architecture .................................................. 18
3.3 The PWC Approach .............................................. 20
  3.3.1 Detecting Indicators of Infection .......................... 20
    3.3.1.1 Indicators and Active Containment .................. 20
    3.3.1.2 Indicator Propagation ................................. 22
    3.3.1.3 Reducing False Indicators ........................... 23
  3.3.2 False-Containment Avoidance ............................... 24
    3.3.2.1 The Vulnerability Window Analysis .................... 24
    3.3.2.2 Traffic Filter for Vulnerability Window Analysis .... 26
  3.3.3 How We Contain a Host .................................... 27
  3.3.4 Containment Relaxation Analysis .......................... 29
    3.3.4.1 When to Relax Containment ............................ 29
    3.3.4.2 Effect of Incorrect Relaxation ......................... 30
3.4 Experiment Setup ............................................... 31
3.5 Evaluation ...................................................... 33
  3.5.1 Tuning Parameters .......................................... 33
    3.5.1.1 The Indicator Threshold ............................... 33
    3.5.1.2 The Size of the Outbound Contact History ........... 33
    3.5.1.3 The Relaxation Analysis Duration ...................... 34
  3.5.2 Performance Evaluation .................................... 35
    3.5.2.1 Containment Performance ............................... 36
      3.5.2.1.1 M1 (Local-to-Local Infection Rate) ............. 36
      3.5.2.1.2 M2 (Escaped Worm Scans) .......................... 37
    3.5.2.2 Impact on Uninfected Hosts ............................ 38
      3.5.2.2.1 M3 (Total Denial-of-Service Time Per Host) ... 39
  3.5.3 Impact of Indicator Propagation Delay ..................... 40
  3.5.4 Impact of Partial Deployment of PWC ....................... 42
  3.5.5 Impact on P2P Traffic ..................................... 43
3.6 Security Analysis ............................................... 44
  3.6.1 Indicator Injection Attacks ................................ 44
    3.6.1.1 Indicator Injection from an External Host ............ 44
    3.6.1.2 Indicator Injection from an Internal Host ............ 45
      3.6.1.2.1 The First Attempt ................................. 45
      3.6.1.2.2 The Second Attempt ............................... 45
  3.6.2 False Positive Indicators .................................. 46
  3.6.3 Worms May Bypass/Disable the Agent after Compromising
      the Host ...................................................... 47
    3.6.3.1 Bypass Attempts ....................................... 47
    3.6.3.2 Disable Attempts ...................................... 47
3.6.4 Other Counterattacks

3.6.4.1 Poisoning Attacks

3.6.4.2 Replacement Attacks

3.6.4.3 Wait-before-Scan Attack

3.7 Discussions

3.7.1 Applicability

3.7.2 Limitations

3.8 Conclusion

Chapter 4 Value Based Software Plagiarism Detection

4.1 Introduction

4.2 Do Core-Values Exist?

4.3 Software Plagiarism Detection

4.3.1 Value Sequence Extraction

4.3.2 Value Sequence Refinement

4.3.2.1 Sequential Refinement

4.3.2.2 Optimization-Based Refinement

4.3.2.3 Address Removal

4.3.3 Similarity Metric

4.3.4 Design Overview

4.3.5 Addressing Reordering Attacks

4.3.5.1 Path Containment Test

4.3.5.2 Path Selection

4.3.5.3 VDG Reduction

4.3.5.4 VDG similarity metric

4.4 Implementation

4.5 Experiment

4.5.1 Case Study I: Obfuscation Tools

4.5.1.1 Impact of Single Obfuscation

4.5.1.2 Impact of Multiple Obfuscation

4.5.2 Case Study II: Similar Programs

4.5.3 Case Study III: Different Programs

4.5.4 Evaluation of the VDG Based Method

4.6 Discussion

4.6.1 Obfuscation Transformations and Attacks

4.6.1.1 Data Transformation

4.6.1.2 Noise Injection (Inserting Arbitrary Instructions)

4.6.1.3 Loop Rewriting

4.6.2 Core-Part Plagiarism

4.6.3 Limitations
List of Figures

3.1 The enterprise network protected by PWC. ........................................ 18
3.2 Time-line at each host running the PWC agent. Alerts from the conventional worm detectors are raised within the range $r_{alert}$. ... 19
3.3 State diagram of a PWC agent monitoring a host. ......................... 21
3.4 Vulnerability Window ............................................................... 24
3.5 $P[N=0]$, the probability that no outbound SYN-ACK and inbound UDP appear within a randomly selected vulnerability window. ... 27
3.6 The active containment rate ($r_{AC}$) and the passive containment rate ($r_{PC}$) for different number of scan suspects ($N$). ............. 34
3.7 Total time spent under containment at each uninfected host in BG1 trace. The maximum length of a single containment was equal to $t_{relax}$ in each case. ................................................................. 35
3.8 The number of infected hosts for local-preferential scanning worms with various scan rates. $V = 6,500, nI_0 = 10$ ......................... 36
3.9 The number of infected hosts for different background traffic (BG1 to 6). $V = 6,500, nI_0 = 10$ ................................................. 37
3.10 The number of escaped scans for different worm’s scan rate. $V = 6,500, nI_0 = 10$ ......................................................... 38
3.11 The number of escaped scans for different background traffic (BG1 to 6). $V = 6,500, nI_0 = 10$ ................................................. 39
3.12 Sorted $r_D$ (average delay per connection request) observed at each host. ................................................................. 40
3.13 Indicator propagation effect in terms of the number of infected hosts. $V = 6,500, nI_0 = 10$ ................................. 41
3.14 Indicator propagation effect in terms of the number of escaped scans. $V = 6,500, nI_0 = 10$ ................................. 41
3.15 The number of infected hosts for different deployment ratio (in log-scale). $r_S = 25$ scans/sec, $V = 6,500, nI_0 = 10$ ........... 42
3.16 The number of escaped scans for different deployment ratio (in log-scale). $r_S = 25$ scans/sec, $V = 6,500, nI_0 = 10$ ........... 43
3.17 Windows Network Architecture Diagram .................................. 46

4.1 Control Flow Flattening Example ............................................. 56
4.2 Sequential refinement example (EAX is initially tainted) ............... 60
4.3 Optimization-based refinement on a plaintiff program .................. 60
4.4 Plagiarism detection process .................................................. 62
4.5 VDG based detection overview ............................................... 64
4.6 Performance ........................................................................... 69
4.7 Similarity scores (y-axis) of original JLex to obfuscated ones and other programs (x-axis) .................................................. 70
4.8 VDG-based similarity scores (y-axis) of original JLex compared to obfuscated ones and other programs (x-axis). Top 10% longest paths are compared .................................................. 75
4.9 Data Transformation Examples. Underlined codes are added by the transformation .................................................. 77
# List of Tables

3.1 Notations used in evaluation. .......................................................... 32

4.1 Proportion of refined value sequences of GCC compiled executables
that overlap value sequences of TCC and WCC compiled executables. 54

4.2 Proportion of refined value sequences that overlap value sequences
of executables obfuscated by Thicket and control flow flattening . 56

4.3 Applicability of value sequence refinement techniques. ............... 59

4.4 Names of obfuscation techniques applied to JLex to generate multi-
    ply obfuscated versions ............................................................. 72

4.5 Similarity scores of four XML parsers cross compared. (P=Plaintiff,
    S=Suspect) ............................................................................. 73

4.6 VDG-based similarity scores of four XML parsers cross compared.
    Top 10% longest paths are compared. (P=Plaintiff, S=Suspect) . 75

4.7 Number of edges and elapsed time of the VDG based method com-
    paring GCC versions. ................................................................. 76
Acknowledgments

I would thank Dr. Peng Liu, Dr. Sencun Zhu, and Dr. Dinghao Wu for inspiring me and supporting me during my doctoral study. They have always been great mentors when I needed either academic or non-academic advices. Also, I would give thanks to all the colleagues who worked with me on the studies presented in this dissertation, especially to Dr. Xiaoqi Jia and Dr. Xinran Wang for their contribution to VaPD. Besides, I would thank Semantic Designs, Inc. and Zelix Pty Ltd. for donating Thicket™ C/C++ Obfuscator and Zelix KlassMaster™ for this study. Thanks to Dr. Jonas Maebe for helping us compiling and running Loco and Diablo. Finally, I would thank God and my family, especially my father, mother, and grandmother for their prayers.
Chapter 1

Introduction

The computer program is a sequence of code through which a programmer directs a computer to do certain tasks. Code can be used for malicious purposes, depending on what the code does or the author’s intention: Code written to exploit a security hole can be used in computer worms and viruses; code used without acquiring proper permission could conduct intellectual property theft; or code inserted with malicious intention may be abused as spyware, botnet, or backdoors to cause many additional security problems (i.e., personal identity theft). As Internet became popular, obtaining source code and technical information on system programming became much easier than before. As a result, the possibility of misuse of code became higher. Also, as everyday life and many aspects of social and economic activities move on to networks, the extent of damage that could be caused by the code written and/or used with malicious intention could be even broader. Therefore, computer security has never been more important than it is these days.

In this study, we consider two types of security issues that can be caused by a non-legitimate use of program code. First, we investigate malicious code. In specific, we propose a technique to quickly suppress propagation of computer worms in an enterprise network. A computer worm exploits security holes of a target machine and implants itself into the computer. Once it infects a host, it starts discovering other victims to replicate itself. A computer worm is written often for malicious purposes (i.e., identity theft, information leakage, denial-of-service attacks, data removal, and immobilizing the host, etc.). However, even a naive worm that simply replicates itself may cause physical damages to the host
and the network (both LAN and WAN) because of the excessive usage of resources (i.e., network capacity, CPU power, etc.) during the aggressive self-replication process. In modern society, many organizations ranging from home to government organizations connect to fast growing networks. Therefore, one can easily imagine the damage that a computer worm might cause would be diverse (from financial loss to national security hazard) and huge. Such a severe worm attack can occur any time under the discretion of the adversary (e.g., hackers, criminals, and terrorists), therefore the risk of worm attacks will not be significantly reduced until highly-effective and highly-practical worm detection and containment technologies are developed and deployed.

Second, we approach to the security issues of codes from a little different point of view: Here we want to propose a practical method to detect illegal use or theft of program code. Plagiarized code itself may not be malicious, but the act of stealing code can be done with malicious or impure motives that are often accomplished by many measures that break the security. In detail, a programmer can learn the techniques used in a program by reviewing source code of the program. Because copying existing code written by others can save a great amount of cost required to develop a program from the scratch, source codes of commercial software are physically protected or protected under a license that provides extremely limited re-usability. Even if the source code is open to public or one has legitimate (often limited) access to the source code, use of the source code in a way violating the license causes intellectual property infringement problems. Blooming of open-source projects accelerated growth of software industry, however, thanks to the open-source projects stealing code also became easier. Without proper measures to detect and regulate software plagiarism, the quantitative growth will against morality of software industry. For the programs of which the source codes are not publicly accessible, malicious programmers can attempt to break software security (i.e., through cracking, hacking, reverse engineering, etc.), network security (i.e., breaking into an enterprise network), or physical security (i.e., spying, bribing, etc.) to illegally access protected information such as source code. Therefore, in a sense, absence of effective methods to detect fraudulent use of program code has a negative influence to security.
1.1 Problems in This Study

In this study, we investigate two crucial security techniques against non-legitimate use of code. First is the fast containment of computer worms spreading in an enterprise network, and the second is the detection of plagiarized code in executable files. These two techniques are used to protect the network and the software industry from security threats discussed above. These techniques are crucial in modern computing environment to provide secure and uninterrupted communications and to promote a more healthy and trustworthy environment for the software industry.

1.1.1 Defense against Propagation of Computer Worms

Typically, worm containment techniques on networks practice a framework of beginning of worm activity; detection completion; and regulation start. When a worm containment system detects a host highly likely to be infected by a worm (or a signature of highly suspicious worm traffic in signature-based systems), it completes detection and starts regulation or containment. For a worm to survive the regulation of worm containment systems, the spread of the worm must be faster than the reaction of containment systems. Therefore, a computer worm tends to make every effort to infect as many victims as possible before the worm containment system starts containment. For example, to evade signature-based worm containment systems, polymorphic/metamorphic worms can change their payload signatures by applying code obfuscation and/or encryption techniques at every hop. Moreover, to evade worm containment systems leveraging certain behavioral characteristics caused by worm’s scanning activities, a worm can adopt different scanning strategies such as local preferential scanning, hit-list scanning, etc. Therefore, an effective worm containment technique must (1) detect potential worm activities in a very early stage of worm’s spread, (2) minimize the amount of time between the start of detection and the start of regulation, (3) be able to deal with detection or containment evasion attempts, and (4) be capable of handling different scanning strategies. In addition, the denial-of-service effect that could be caused by false positive detection of worm activities must be insignificant.
1.1.2 Defense against Code Plagiarism

We have some technical issues and realistic issues in software plagiarism detection area. A typical and the most straightforward way to determine code theft is to analyze the source code. However, the plagiarizer can make the source code extremely difficult to read through systematic application of code obfuscation techniques. A code obfuscation technique transforms the syntactic structure of a program into a completely different one while preserving only the semantic structure of the program. Therefore, code obfuscation techniques can make it exceedingly arduous for human (or for traditional code comparison or clone detection techniques) to compare the original program to the obfuscation output. Moreover, in typical real world situations, one must find significant evidences of actual plagiarism to obtain access to the source code of a commercial program of which the source code is not open to public. Finally, to be practical, a software plagiarism detection technique should not be specific to certain programming languages or operating systems. So, an effective and practical technique to detect software plagiarism must (1) be applicable directly to executable files without accessing suspicious program’s source code, (2) minimize the influence of code obfuscation techniques in detection, and (3) be platform independent.

1.2 Contributions of This Study

1.2.1 Worm Containment

The proposed worm containment technique (namely Proactive Worm Containment System, a.k.a. PWC) develops a new framework of worm containment: beginning of worm activity, worm infection indicator detection, regulation start, relaxation analysis, and selective relaxation of mis-contained hosts. The infection indicator can be available much earlier than the detection result (i.e., signatures and filters) of traditional worm containment systems. Therefore, PWC can reduce the number of hosts that could be newly infected during the detection delay between the beginning of worm activity and the regulation start of traditional worm containment framework. PWC attempts to prevent unnecessary containment by applying the white-host detection technique which can determine whether the host needs to be
contained or not, immediately before starting containment. To further mitigate possible availability loss, PWC performs the relaxation analysis immediately after starting containment, and relaxes mistakenly contained hosts quickly after the analysis. Because PWC does not depend on worm signatures, it does not miss polymorphic/metamorphic worms. Also, PWC flexibly handles computer worms practicing different scanning strategies. For example, by minimizing the number of failed scans, hit-list based worms can evade worm containment systems leveraging ICMP type 3 “host unreachable” message to detect scanning activities. PWC will not be affected by such worms because it uses one of the most fundamental characteristics of worm’s scanning activities.

1.2.2 Software Plagiarism Detection

Proposed technique (namely Value-Based Software Plagiarism Detection, a.k.a. VaPD) develops a novel semantic indicator of run-time behavior of a program. In a series of pilot experiments, we observe that some of the runtime output values of machine instructions are extremely hard to be replaced or eliminated through semantics-preserving transformations. VaPD extracts such invariant runtime values of given program directly from the executable through a virtual machine based technique without accessing the source code. From the extracted values, VaPD constructs a value sequence showing the sequential order in which the values are computed during execution, and significant overlapping of the value sequences of two programs indicates potential plagiarism. Moreover, the value sequence can be constructed as value dependence graph where the nodes represent the values and the edges represent dependence between the values. Each path of a value dependence graph shows unchangeable partial ordering of values, therefore it is robust against the evasion attempt in which plagiarizers reorder the values in a way that reduces the overlapping subsequence below the detection threshold. To our knowledge, VaPD is the first system that uses runtime values to detect software plagiarism. It is resilient to code obfuscation techniques: for all the code obfuscation techniques (more than 38 techniques) that we have tested in this research both individually and in combination, VaPD successfully identified the original program for each obfuscated program. We believe this is because VaPD uses one of the most
fundamental run-time characteristics of a program. Since VaPD does not require a plaintiff to obtain the access to source code of a suspect program, VaPD could greatly reduce plaintiff’s risks through providing strong evidence before filing a lawsuit related to intellectual property. In addition, VaPD is fairly practical. It can directly work on binary executables of the suspect program, and its effectiveness is not restricted to whole-program plagiarism. Although core-part plagiarism detection is still an open problem that requires much future research, VaPD can be used to solve certain categories of core-part plagiarism detection problems. Finally, as a generic runtime characterization technique, VaPD’s potentials are not limited to plagiarism detection. For example, VaPD is also applicable for identifying and classifying malcode variants that are obfuscated through metamorphism and/or polymorphism.

1.3 Outline

The thesis is organized as follows. In Chapter 2, we present the main researches related to our studies. In Chapter 3, we propose the proactive worm containment technique for enterprise networks. In Chapter 4, we propose the value-based software plagiarism detection technique. In Chapter 5, we summarize the thesis, and overview the future work.

Chapter 3 is organized as follows. In Section 3.2, we briefly explain overall architecture of our system following time-line and, in the next section, we illustrate key operations of our system one by one. Each of the subsections explains one of the operations of PWC system. In Section 3.4 and Section 3.5, our system is evaluated in comparison with Hamsa and Williamson’s Virus Throttle. Section 3.6 takes up security issues on our system, and the next section deals with applicability and limitation. Conclusion summarizes the new worm containment approach introduced in this study.

Chapter 4 is organized as follows. Section 4.2 presents the concept of core-values, the runtime values that are invariant through the semantics-preserving transformations. Section 4.3 takes up details of extraction and refinement of value sequences, and we put it together in Section 4.3.4. In Section 4.3.5, we discuss about the reordering attempts to evade the value sequence based detection, and
introduce the value dependence graph based detection to address this issue. Section 4.4 deals with implementation issues. Experiment results and discussions are presented in Section 4.5 and Section 4.6 respectively. Section 4.7 concludes our study on software plagiarism detection.
Chapter 2

Related Work

In this chapter, we summarize related work in the literature, and give an overview on the different perspectives these researches have addressed. However, the detail discussion on the most related approaches and the rationale on why we need new ideas are presented in the corresponding chapters.

2.1 Worm Containment

Existing worm containment techniques can be roughly broken down into five classes as follows.

2.1.1 Class A: Rate limiting

The idea of Class A techniques is to limit the sending rate of scan-like traffic at an infected host. Virus Throttle proposed by Williamson et al. [1] uses a working set and a delay queue to limit the number of new machines that a host can connect to within unit time. In [2], connection failure rate is exploited, and, in [3], the number of unique IP addresses that a host can scan during each containment cycle is leveraged. In [4], various deployment strategies of rate control on the Internet are compared. Zou et al. [5], to decelerate a worm epidemic, contain a host for a predefined duration (e.g., a couple of seconds) upon detection of any suspicious behavior committed by the host. Class A techniques may introduce longer delays for normal traffic. They may not effectively stop a worm especially with high
chances of hitting actual victims (e.g., local preferential scanning worms with large vulnerable population). Although PWC and [3] (an independent work) do free-of-infection checking (called white-host detection in PWC), PWC is very different from [3] in that significant denial-of-service could be possible in a heavy duty checking process “assumed” in [3]. PWC does automated, lightweight, efficient white-host detection.

2.1.2 Class B: Signature-based worm scan filtering

The idea is to generate the worm signature that can then be used to prevent scans from entering or leaving a LAN or a host. Earlybird [6] and Autograph [7] are efficient inline solutions that integrate flow classification, signature generation and scan filtering. However, they can be easily evaded by obfuscation techniques such as polymorphism. Polygraph [8] can handle polymorphic worms, but it spends too much time in analyzing live traffic and generating the signature. In [9, 10, 11], signatures are generated out of packets “captured” by a honeypot. However, network-level flow classification techniques used invariably suffer from false positives leading to noise in the worm traffic pool [12]. Although Hamsa [12] is a fast, noise-tolerant solution against malicious network flows, the false negatives and false positives of a signature depend on the accuracy of the flow classifier used. In addition, Hamsa and many other Class B solutions are vulnerable to Polymorphic Blending attacks [13]. COVERS [14] is a fast, host-level signature generator leveraging code running symptoms of worms. However, COVERS is an application specific technique that requires special auditing/diagnosis facilities. PWC is signature-free - containing worms without using signatures. Accordingly, although PWC cannot prevent worm scans from entering a host, PWC is in general much more resilient to polymorphic worms and worm code obfuscation than Class B techniques, and PWC has much better timeliness.

2.1.3 Class C: Filter-based worm containment

Class C techniques share the same spirit with Class B techniques except that a filter is a code fragment that can be used to check if a given input message would cause abnormal behavior of target application. Shield [15] uses host-based filters
to block vulnerabilities but these filters are generated manually. Vigilante [16] generates and distributes host-based filters automatically. But, its response time relies on the worm payload size, and some filters can be evaded by code obfuscation based on character shifting or insertion. To achieve high coverage, they need a complicated detection technique such as dynamic dataflow analysis [17, 18]. PWC does not use such filters.

2.1.4 Class D: Payload-classification based worm containment

The idea of Class D techniques is to determine if a packet contains a worm. In [19, 20, 21], several anomaly detection techniques are proposed to detect worms. But, they may suffer from significant false negatives or false positives, especially in the presence of code obfuscation. In [22], control flow structures are exploited to detect polymorphic worms, but off-line analysis is required. In [23, 24], they detect whether a data packet contains code, but not all worms propagate through data packets. PWC does not perform code analysis on payloads.

2.1.5 Class E: Threshold Random Walk (TRW) scan detection

TRW [25] exploits randomness in picking destinations to connect to, to detect whether a host is a scanner. In [26], a network-based hardware implementation is investigated. TRW is suitable for deployment in high-speed, low-cost network hardware, and it is very effective in tackling the common way of worm scanning (i.e., random scanning with high failing likelihood). Attackers may evade TRW, using such attacks as two-sided evasion [26] to which PWC is not vulnerable.

2.1.6 Other related work

(a) In [27, 28], such macro symptoms as Internet background radiation are observed by network telescopes to raise early warnings of Internet-wide worm infection. Such warnings, though not required, may help PWC to confirm its judgment of LAN worm infection. (b) Address-space randomization [29] and instruction-set
randomization [30, 31] may prevent a host from being infected, but they may result in termination (or crashing) of exploited processes, which effectively renders the service unavailable. (c) Modifying some aspects of the OS [32, 33, 34] or the processor [35] may prevent buffer overflows, but these approaches require existing software or hardware components to be changed.

2.2 Software Plagiarism Detection

2.3 State of the Art

We roughly group the literature into three categories: code obfuscation techniques, static analysis based plagiarism detection, and dynamic analysis based plagiarism detection.

Code Obfuscation Techniques: Code obfuscation is a semantics-preserving transformation to hinder figuring out the original form of the resulting code. A generic code obfuscation technique is not as simple as adding $x$ before computation and subtracting $x$ after the computation. Collberg et al. [36] provided an extensive discussion on automated code obfuscation techniques. They classify code obfuscation techniques in the following categories depending on the feature that each technique targets: data obfuscation, control obfuscation, layout obfuscation, and preventive transformations. Collberg et al. also introduced Opaque Predicates [37] to thwart static disassembly. Other techniques such as indirect branches, control-flow flattening, and function-pointer aliasing were introduced by Wang [38].

Several code obfuscation tools are available. SandMark [39] is one of such tools implementing 39 obfuscators applicable to Java bytecode. Array representation and orientation, functions, in-memory representation of variables, order of instructions, and control and data dependence are just a small set of the features that SandMark can alter. Another Java obfuscator is Zelix KlassMaster [40]. It implements comprehensive flow obfuscation techniques, making it a heavy-duty obfuscator. Semantics is the only characteristic guaranteed to be preserved across the obfuscation.

Static Analysis Based Plagiarism Detection: The existing static analysis techniques except for the birthmark-based techniques are closely related to the
While possessing common interests with the clone detection, the plagiarism detection is different in that (1) we must deal with code obfuscation techniques that are often employed with a malicious intention; (2) source code analysis of the suspicious program is not possible in most cases. Static analysis techniques for software plagiarism detection can be classified into five categories: string-based [41], AST-based [50, 51, 52], token-based [53, 54, 55], PDG-based [56], and birthmark-based [57, 58].

**String-based:** Each line of source code is considered as a string. A code fragment is labeled as plagiarism if the corresponding sequence of strings matches certain code fragment from original program. **AST-based:** The abstract syntax trees (AST) are constructed from two programs. If the two ASTs have common subtrees, plagiarism may exist. **Token-based:** A program is first parsed to a sequence of tokens. The sequences of tokens are then compared to find plagiarism. **PDG-based:** A program dependency graph (PDG) represents the control flow and data flow relations between the statements in a program procedure. To find plagiarism, two PDGs are constructed and compared to find a relaxed subgraph isomorphism. **Birthmark-based:** A software birthmark is a unique characteristic of a program that can be used to determine the program’s identity. Two birthmarks are extracted from two programs and compared.

None of the above techniques is resilient to code obfuscation. String-based schemes are vulnerable even to simple identifier renaming. AST-based schemes are resilient to identifier renaming, but weak against statement reordering and control replacement. Token-based schemes are weak against junk code insertion and statement reordering. Because PDGs contain semantic information of programs, PDG-based schemes are more robust than the other three types of the existing schemes. However, the PDG-based methods are still vulnerable to many semantics-preserving transformations such as inlining/outlining functions and opaque predicates. The existing birthmark-based schemes are vulnerable to either obfuscation techniques mentioned in [57] or some well-known obfuscation such as statement reordering and junk instruction insertion. Moreover, all existing techniques except for [57, 49] need to access source code.

**Dynamic Analysis Based Plagiarism Detection:** Myles and Collberg [59] proposed a whole program path (WPP) based dynamic birthmark. WPP was orig-
inally used to represent the dynamic control flow of a program. WPP birthmarks are robust to some control flow obfuscation such as opaque predicates insertion, but are still vulnerable to many semantics-preserving transformations such as flattening and loop unwinding. Tamada et al. [60, 61] also introduced two types of dynamic birthmarks for Windows applications: Sequence of API Function Calls Birthmark (EXESEQ) and Frequency of API Function Calls Birthmark (EXEFREQ). In EXESEQ, the sequence of Windows API calls is recorded during the execution of a program. These sequences are directly compared to find the similarity. In EXEFREQ, the frequency of each Windows API call is recorded during the execution of a program. The frequency distribution is used as a birthmark. Schuler et al. [62] proposed a dynamic birthmark for Java. The call sequences to Java standard API are recorded and the short sequences at object level are used as a birthmark. Their experiments showed that their API birthmarks are more robust to obfuscation than WPP birthmarks. These birthmarks, however, can only identify the same source code compiled by different compilers with different options, and the performance against real obfuscation techniques is questionable. For example, attackers may simply embed some of API implementations into their program so that fewer API calls will be observed. Wang et al. [63] proposed a system call based birthmark, addressing the problems with API based techniques. However, the proposed technique cannot be applied to computation oriented software containing few system calls, and is still vulnerable to injecting transparent system calls in the middle of an edge on the system call dependence graph.
Chapter 3

PWC: A Proactive Worm Containment Solution for Enterprise Networks

3.1 Introduction

Computer worms (i.e., malicious self-propagating code) are a significant threat to Internet security. The severity of their damage has been well demonstrated by a set of high-profile Internet wide worm attacks: (a) Within merely 5 minutes, 75,000 SQL Servers were infected by the Slammer worm in 2003: the peak global scan rate was above 55 million scan packets per second; a lot of enterprise networks were fatally congested; and many sites had to be down for recovery. (b) Within 16 hours, about 350,000 hosts were infected by the CodeRed-II worm in 2001. (c) In 2004, the Witty worm infected about 12,000 firewall systems where data stored in certain portions of the disks were destroyed. These high-profile worm attacks have clearly shown that the loss caused by a worm attack against an enterprise (and her business) can be potentially huge (e.g., tens of millions of dollars). Such a severe worm attack can occur at literally any point of time under the discretion of the adversary (e.g., hackers, criminals, and terrorists), therefore the risk of worm attacks will not be significantly reduced until highly-effective and highly-practical worm detection and containment technologies are developed and deployed.

Since worm infection can spread more rapidly than human response, automated
worm detection and containment techniques are essential. In addition, worm containment techniques should be able to handle zero-day (unknown) worms. Enterprise level worm containment has three basic goals: (1) prevent internal hosts from being infected; (2) block outgoing worm scans; (3) minimize the denial-of-service effects caused by worm containment controls.

Many approaches have been proposed to perform the enterprise level worm containment. However, current defenses do not complete four specific requirements which include (R1) timeliness in policing worm scans, (R2) resiliency to containment evading, (R3) minimal denial-of-service costs, and (R4) being agnostic to worms’ scanning strategy to contain a wide spectrum of worms from uniformly randomly scanning worms to topologically aware scanning worms. [6, 7, 8, 12, 14, 16] have limited application of R1, [6, 7, 8, 12, 14, 16, 19, 20, 22, 24, 26] are short of R2, [1] lacks R3, and [7, 8, 12, 16, 24, 26] have limitation in R4.

To overcome above limitations, we propose PWC, a novel proactive worm containment solution for enterprises. The idea of PWC is motivated by two important observations: (O1) If every infected host can be immediately disabled from releasing UDP packets or getting outgoing TCP connections connected, the worm will be contained, even if incoming UDP and TCP connections are still allowed. (O2) In order for a worm to be fast in propagating itself, any infected host must use a sustained faster-than-normal outgoing packet rate.

O1 and O2 indicate that PWC may use a sustained faster-than-normal outgoing connection rate to be aware that a host is infected and the awareness can be gained many seconds before a signature or filter is generated; then the host’s outgoing UDP packets and TCP connection attempts can be instantly blocked – instead of being slowed down – to achieve quick, proactive containment. By providing novel ways to handle false positives transparently or with minimized impact, PWC allows worm containment systems to react earlier without suffering from high false-positive rates.

To overcome denial-of-service effect that could be caused by false positives (in identifying infected hosts), PWC develops following two novel white-host detection techniques: (a) PWC exploits a unique vulnerability window lemma to avoid false initial containment; (b) PWC uses a relaxation analysis to uncontain (or unblock) those mistakenly contained (or blocked) hosts within few seconds, if there
are any. Note that existing rate-limiting techniques are worm-slowing-down tech-
niques, while PWC is a *worm-halting* technique. Finally, PWC integrates itself
seamlessly with existing signature-based or filter-based worm scan filtering solu-
tions. As soon as a signature or filter is generated, PWC can stop enforcing any
new containment controls and unblock all still-being-contained hosts. With a lit-
tle extension, PWC can utilize the multi-resolution worm detection technique [64]
and react against slower worms. In this study, we will discuss single resolution
detection as our focus is on *contain-and-relax* framework.

We have evaluated the cost-effectiveness of PWC using both real world traces
and extensive simulation experiments. Our empirical study shows that PWC is
significantly outperforming Virus Throttle scheme proposed by Williamson *et al.*
[1] in terms of all of the three evaluation metrics: (M1) number of released worm
scans, (M2) number of hosts infected by local worm scans, (M3) total denial-of-
service time per host. Moreover, the experiments show that PWC is significantly
outperforming Hamsa [12] in terms of M1 and M2 with negligible denial-of-service
costs. The merits of PWC summarized below show that PWC has taken a major
step forward in meeting the requirements aforementioned.

⊙ PWC is *signature-free*; it does not rely on worm signatures. Without the need to
match a message or payload with a signature or a filter, PWC can *defend against
polymorphic worms and all worm code obfuscation methods*.
⊙ Exploiting an obvious property of scanning worms (i.e., any infected host must
use a sustained faster-than-normal outgoing packet rate), PWC is *resilient to con-
tainment evading*.
⊙ *Timeliness.* PWC may react to worm scans many seconds before a signature or
filter is generated.
⊙ PWC is *agnostic* to the scanning strategy of worms since it does not rely on
any symptoms caused by specific scanning strategy (i.e., ICMP type 3 messages
for failed scans).
⊙ Exploiting the vulnerability time window theorem and the white-host detection
idea, PWC causes *minimal* denial-of-service.
⊙ PWC is NOT protocol specific.
⊙ PWC performs containment consistently over a large range of worm scan rates.
PWC is *not sensitive* to worm scan rate.
In the next section, we describe related work categorized into five classes. In Section 3.2, we briefly explain overall architecture of our system following time-line and, in the next section, we illustrate key operations of our system one by one. Each of the subsections explains one of the operations of PWC system. In Section 3.4 and Section 3.5, our system is evaluated in comparison with Hamsa and Williamson’s Virus Throttle. Section 3.6 takes up security issues on our system, and the next section deals with applicability and limitation. Conclusion summarizes the new worm containment approach introduced in this study, and remaining work.

3.2 PWC Overview

3.2.1 Definition and Scope

Target Worm. We consider UDP/TCP-based scanning worms. We also consider hit-list scanning worms and topologically aware worms including local preferential scanning worms.

Worm Scan. We classify worm scans in three types: L-L scans from an internal (local) infectee to an internal address, L-R scans from an internal infectee to an external (remote) address, and R-L scans from an external infectee to an internal address.

Local Host of a Worm Agent. PWC is based on collaborating worm agents (a.k.a. PWC agents) each of which is associated with a host in the protected network. A PWC agent can be a part of an operating system, or an off-the-box implementation (e.g., firmware in NICs and switches). A host assigned to a PWC agent is referred as local host of the PWC agent, and the PWC agent is referred as local PWC agent of the host.

Connection Attempts and Connections. An outbound connection attempt is defined as either an outbound TCP SYN or an outbound UDP packet. A successful outbound connection is defined as an observation of an inbound TCP SYN-ACK packet. UDP packets are always considered as successful connections. Inbound connections/connection attempts are defined in similar ways. When we mean TCP connections, we use ‘TCP connections’ explicitly.

Signature Extractors. PWC can be a layer of defense in a multi-layer defense
approach. As an example of another layer, we assume an automatic signature generation system(s) is operational in the same enterprise network.

### 3.2.2 Architecture

As shown in Figure 3.1(a), each host in the protected enterprise network runs a PWC agent that performs detection and suppression of worm scans released from its host. We conceptualize a host with a PWC agent as shown in Figure 3.1(b). Discussions on implementation issues can be found in Section 3.6.3. The PWC manager has two roles: first, it distributes authenticated worm alert reported by a PWC agent to all the other PWC agents in the enterprise network; second, it is a certificate authority in authentication of propagated alerts between each PWC agent and the PWC manager and vice versa. PWC can handle multiple simultaneous alerts raised by different worms in one contain/relax procedure.

Before getting into details, we briefly summarize operations of PWC system, from A to G, in an event-driven manner following the time line in Figure 3.2. We will discuss the details on following operations in Section 3.3 one by one.

1. **When a PWC agent detects a possible scan activity.** The PWC agent takes
The agent detects scanning activity.
- Contain the host
- Create/propagate an indicator
- Begin the relaxation analysis

Relaxation failures

Isolate

time

Infection

(a) Active containment

Figure 3.2. Time-line at each host running the PWC agent. Alerts from the conventional worm detectors are raised within the range $r_{alert}$.

The following actions in order: (a) raises an early-but-immature worm alert or an indicator; (b) initiates containment on its host, which is called *active containment* as in Figure 3.2(a); (c) reports the indicator to the PWC manager; (d) starts *relaxation analysis* operations on its host. c is to let other PWC agents be aware of the situation and check their hosts if they are infected. d is required since the agent needs to detect sustained rate of connection attempts to distinct destination addresses, to determine the host is infected.

(2) When *PWC manager receives an indicator*. The PWC manager propagates the indicator to all other agents. Note that the focus of this study is not on the underlying mechanism of the indicator propagation. The frequency of indicator propagation is controlled by PWC.

(3) When a *PWC agent receives an indicator*. The PWC agent takes following actions in order: (a) it performs *vulnerability window analysis* (see Section IV) to see whether its host is possibly infected or not; (b) if no evidence of possible infection is found, it ignores the indicator; (c) otherwise, it initiates containment on its host, which is called *passive containment* as in Figure 3.2(b); (d) and it immediately starts *relaxation analysis*. a and b minimize availability-loss possibly
cased by excessive passive containments.

(4) *When a PWC agent is performing relaxation analysis.* During the relaxation analysis, the PWC agent calculates the rate of outbound connection attempts to distinct IP addresses, and checks if the host shows sustained connection rate or not. The relaxation analysis is limited to $t_{relax}$ seconds.

(5) *When a PWC agent completes relaxation analysis.* Based on the result of (4), the agent relaxes or continues the containment. If the agent relaxes the containment, it will repeat above operations from (1). If the agent continues the containment, it will repeat (4) once more. After $F$ relaxation failures, the agent will isolate its host and report to the PWC manager for further handling. We observed no isolated uninfected host through number of experiments with $F = 30$ times and $t_{relax} = 1$ second.

(6) *When signature extractors identify new signatures.* The signatures are reported to the PWC manager.

(7) *When PWC manager receives a signature.* The PWC manager sends the signature to a security manager so that it may install the signature into firewalls to block inbound (or outbound) malicious messages. At the same time, the signature is propagated to all PWC agents and will be installed in the agents’ embedded packet filters. The packet filter reduces unnecessary propagation of indicators by filtering out previously identified malicious messages.

### 3.3 The PWC Approach

PWC consists of three major phases: *detecting indicators of infection* (Section 3.3.1), *initial containment* (Section 3.3.2, 3.3.3), and *relaxation* (Section 3.3.4) phases. In this section, we will illustrate each of them in order.

#### 3.3.1 Detecting Indicators of Infection

#### 3.3.1.1 Indicators and Active Containment

PWC starts with detecting an *indicator* of infection. The indicator requires to be detected early, but not necessarily to have an extremely low false-positive rate. This important characteristic allows PWC agents to contain possibly infected hosts
swiftly without hesitation while requiring consequent relaxation phases to resolve the false positives. Since, to survive in the wild, a worm must replicate itself to at least another victim before being contained, a worm naturally sends infectious messages to as many distinct destination addresses as it can. Therefore, abnormal growth in the number of distinct addresses at infected hosts has been in the literatures [6, 65, 3]. For fast worms, even a second of monitoring would be sufficient to confirm this attitude. We observed in a 24-hour Auckland-IV trace [66] that majority of the hosts connected to less than 15 distinct IPs/sec, and only few of them connected to 20-25 distinct IPs/sec. In our lab traces, the rates of distinct destination addresses were no more than 5 IPs/sec. In contrast, the CodeRed-I, for example, probes more than a hundred unique IP addresses per second.

Algorithm 1 shows how PWC agents detect indicators and initiate active containment (Figure 3.2(a)). ON_OUT_CONNECTION_CONTAIN() in line 2 follows the description in Section 3.3.3 and Section 3.3.4. A PWC agent considers an outbound connection attempt to a new IP address as a scan suspect. On monitoring every scan suspect going out from a host, the local PWC agent of the host calculates the rate $r$ based on the most recent $N$ elements in outconhist, the outbound contact history which is a list of the time-stamps of recent scan suspects. If $r$ exceeds the threshold $\lambda$, the PWC agent detects an indicator, initiates active
Algorithm 1 ON_OUT_CONNECTION_CONTAIN(pkt)

Require: Known dest IP set dsthist, known src IP set srchist, pkt = packet to send, outconhist = recent $N$ outbound connection attempts.

if host is contained then
    ON_OUT_CONNECTION_CONTAIN(pkt)
    return
end if

if pkt.dst_ip $\in$ dsthist $\cup$ srchist then
    Process pkt
    return
end if

Insert pkt to outconhist

$r \leftarrow \frac{\text{outconhist.size}}{\text{outconhist.size}}$ \text{outconhist}\[\text{latest}.time - \text{outconhist}\[\text{oldest}.time]$

if $r > \lambda$ then
    Start active containment
    Report an infection indicator to PWC manager
end if

containment on its host, and reports the indicator to the PWC manager.

3.3.1.2 Indicator Propagation

Any indicator of infection detected at a host implies the possibility of hidden infectees in the network. To proactively block the hosts that are infected but not detected, PWC agents disseminate an indicator through the indicator propagation. On detection of an indicator, in addition to starting active containment on its local host, a PWC agent reports the indicator to the PWC manager. The PWC manager then broadcasts the indicator to all the agents in the network. Information propagation techniques for cooperative defenses against Internet worms are in the literature [67, 68]. In this study, we assume a technique for the PWC manager to propagate indicators.

The following message carries an indicator over the network: $[t_{\text{sent}} + t_d + \text{the agent’s IP}]$. $t_d$, the detection latency to be used in Section 3.3.2.1, is defined as $t_{\text{sent}} - t_{\text{in}}$, where $t_{\text{sent}}$ is the current time and $t_{\text{in}}$ is the time-stamp of the latest successful inbound connection before the oldest among the $N$ time-stamps referenced in calculating $r$. To prevent possible bandwidth saturation caused by worms from interfering with the indicator report, the agent reports the indicator
after containing its host.

The receivers of either reported or propagated indicators would discard the indicators if $t_{\text{sent}}$ is too old. To prevent forged indicator injection, all the messages between PWC agents and the manager should be authenticated using RSA. Details are discussed in the Security Analysis Section (Section 3.6.1). The behavior of a PWC agent after receiving a propagated indicator is described in Section 3.3.2.

**Algorithm 2 REPORT_INDICATOR($\text{indicator}$)**

- **Require:**
  - $\text{indicator} =$ infection indicator to report,
  - $t_{\text{received}} =$ time of the latest indicator received from the PWC manager,
  - $t_{\text{relax}} =$ relaxation analysis duration,
  - $t_{\text{now}} =$ current time.
- **if** $t_{\text{received}} + t_{\text{relax}} > t_{\text{now}}$ **then**
  - **return**
- **end if**
  - Send signed $\text{indicator}$ to the PWC manager.

As shown in Algorithm 2, a PWC agent tries to control the frequency of indicator propagation in order to avoid denial-of-service and overwhelming traffic. An indicator will not be reported to the PWC manager if the time elapsed since the most recently received indicator is less than the relaxation analysis duration $t_{\text{relax}}$. In this case, we consider all the suspicious hosts in the network are already contained by the previously received indicator and are in the relaxation analysis. The PWC manager also applies similar restriction. Therefore, the indicator propagation rate is limited to $\frac{1}{t_{\text{relax}}}$ times per second.

### 3.3.1.3 Reducing False Indicators

In our network, false positive indicators were mainly caused by the applications that send excessive small UDP packets to many distinct destinations (e.g., P2P file sharing and mDNS protocols). To reduce such false indicators, we ignore outbound UDP packets that are shorter than 200 bytes. A scanning worm inherently has a byte-sequence to exploit vulnerability, code to select victims, code to craft infectious messages, code to send infectious messages, and at least one loop. Polymorphic/metamorphic worms require additional code to obfuscate/restore themselves. Even though a worm may use smaller packets to probe vulnerabilities and longer packets to infect discovered victims, a worm must have a lower limit in size.
Figure 3.4. Vulnerability Window

For example, among the 100 UDP-based worms obtained from Symantec’s Viruses & Risks Search, the smallest payload length was 376 bytes (SQL Slammer). In addition, please see Section 3.5.5 for more discussion on worm-like P2P traffic.

3.3.2 False-Containment Avoidance

A propagated indicator makes every agent in the network start passive containment (Figure 3.2(b)). On receiving a propagated indicator, the agent validates the indicator first, which we named *false-containment avoidance*. Note that passive containment initiated by the propagated indicator is a *proactive* action taken on a host that is not suspicious to local PWC agent’s knowledge. Therefore, any propagated indicator can be ignored when the receiving agent can ensure that its local host is not infected. The decision results in either of *SAFE* and *UNSAFE*, where *SAFE* means the PWC agent can safely ignore the indicator, and UNSAFE means the agent should not.

3.3.2.1 The Vulnerability Window Analysis

Consider PWC is fully deployed in an enterprise network. Let us assume all the PWC agents configured with the same parameters since, typically with many organizations, most hosts within the same enterprise network would have similar ability to send packets. Let us assume that infected host $h_1$ creates an indicator and propagates it through the PWC manager. Given that $h_2$ is one of recipients
of the propagated indicator, let us depict the timeline of the propagation in Figure 3.4 where,

i. $t_1$ at $h_1$ is the time of the last successful inbound connection before releasing the first scan.

ii. $t_2$ at $h_1$ is the time when (potentially) the first scan is released.

iii. $t_0$ at $h_1$ is the time when an indicator is created.

iv. $\Delta t$ is equal to $(t_0 - t_1)$

v. $t'_0$ at $h_2$ is the time of receiving an indicator from $h_1$

vi. $t'_1$ at $h_2$ is equal to $(t'_0 - \Delta t)$

vii. $t_{in}$ at $h_2$ is the time of the last successful inbound connection.

Let us assume (a) $h_2$ is susceptible to the same worm as $h_1$ has; (b) $h_2$ is not contained at $t'_0$; (c) $\Delta t < t_{relax}$; (d) $h_1$ and $h_2$ have similar CPU/NIC performance. (a) and (b) are considered to be true, PWC should be configured to hold (c), (d) is generally true in an enterprise network. We do Vulnerability Window Analysis by testing the following hypothesis: (e) the connection attempt made at $t_{in}$ was infectious. The merit of this analysis is that if the hypothesis is proven False, $h_2$ can safely ignore the indicator and avoid containing an innocent host. To see if the hypothesis is False, we assume the hypothesis were True, then we prove by contradiction.

To determine whether $h_2$ needs to be contained or not at time $t'_0$, we must consider following cases (1) and (2).

(1) $t_{in} < t'_1$: If hypothesis (e) were True, $h_2$ should have been infected at $t_{in}$, and PWC agent at $h_2$ must have detected an indicator within the time window $[t'_1, t'_0]$ and become contained. From (b), $h_2$ is not contained at $t'_0$, thus we can conclude $h_2$ was not infected at $t_{in}$. Because $h_2$ has never been connected since $t_{in}$, $h_2$ is considered to be SAFE.

(2) $t_{in} > t'_1$: $h_2$ should be considered to be UNSAFE, for we cannot reject hypothesis (e).

Therefore, we have Lemma 1, *vulnerability window lemma*. 
Lemma 1. Given that $t_d$ is detection latency of the sender, if $h_2$ receives a propagated indicator $(t_0, t_d, h_1)$ at $t'_0$, $h_2$ can ignore the indicator and skip passive containment if the following assumptions hold:

i. $t_{in} < t'_0 - t_d$

ii. $h_2$ is susceptible to the same worm as $h_1$ has.

iii. $h_2$ is not contained.

Lemma 1 can be extended to handle multiple kinds of worms by taking the larger $t_d$ when indicators report different $t_d$’s. Although a worm can evade passive containment by having a delay before starting scanning, the worm cannot successfully spread out since local PWC agent will initiate active containment after monitoring the first $N$ scans.

A limitation of vulnerability window analysis is that any inbound connection attempt within the vulnerability window makes the vulnerability window analysis result in UNSAFE. The result is affected by two factors: first, frequent legitimate inbound connections; second, large vulnerability window $\Delta t$. We will introduce two heuristics to address these limitations and will see how often the vulnerability window analysis would raise false positives with selected $\Delta t$. From the definition of $t_d$ in Section 3.3.1.2, the largest $\Delta t$ can be approximated as $\frac{N}{\alpha}$ seconds when $\alpha$ is zero. (7,4) and (7,10), the two pairs of $(\lambda, N)$ that we configured based on real trace experiments, yield $\Delta t = 0.57$ and 1.43 seconds respectively.

3.3.2.2 Traffic Filter for Vulnerability Window Analysis

To make the vulnerability window analysis resilient to noise (legitimate traffic), we set up two heuristics to sift out meaningful traffic within the vulnerability window. The heuristics are:

- **H1**: Even an internal worm scanning 8,000 IPs per second with 50% of local preference would take more than 16 seconds to scan entire /16 local network. Therefore, we regard redundant connection attempts from the same IP address incoming within $H_t$ seconds as a noise, and reduce them leaving only the first one.
- **H2**: Eliminate inbound UDP packets whose payloads are shorter than $H_t$ bytes. PWC uses $H_t = 200$ as we discussed in Section 3.3.1.3.

We could reduce 96% of the legitimate inbound connection attempts appeared in our lab PC traces by H1 and H2 with $H_t = 10$ and $H_t = 200$.

A useful tool to find an effective vulnerability window size is $P[N=0]$, the probability that vulnerability window at a certain time point may not include legitimate inbound connection attempts. As shown in Figure 3.5, we calculated $P[N=0]$ on our lab PC traces (PC A/B), sliding the vulnerability window with a granularity of 0.0001 seconds. PC A and PC B were dedicated to two different individuals User-A and User-B respectively. During 7 hours of traffic collection, web browsing was majorly done by User-A and User-B. Besides, User-B was subscribing Internet stock information and video streaming services. $P[N=0]$ could change if user behavior changes as we can see from PC A and PC B traces. Heuristics H1 and H2 also could reduce the difference, yielding overlapping PC A+Filter and PC B+Filter lines.

### 3.3.3 How We Contain a Host

During active or passive containment, a PWC agent prohibits its local host (which is suspected to be infected by a worm) from connecting to other hosts. We designed
containment strategy in PWC to meet the following requirements:

- Containment should cause minimum side effect on legitimate traffic.
- Containment should be able to prevent contained host from actively connecting to internal/external hosts.
- Containment should not interfere with other worm defense systems such as network-based signature identification and filtering techniques.

```
Algorithm 3 ON_OUT_PACKET_CONTAIN(pkt)
Require: inconhist = recent inbound connections, pkt = packet to send.
if pkt is not UDP nor TCP-SYN then
    Forward pkt
    return
end if
if pkt is UDP and pkt.dst_ip ∈ {source addresses in inconhist} then
    Forward pkt
    return
end if
Perform relaxation analysis
Insert pkt into buffer
if outbound connection rate > σ then
    return
end if
if pkt.dst_ip is within the same /16 then
    return
end if
pkt.TTL ← TTL to border
Forward pkt.
```

A PWC agent processes outbound connection attempts initiated during containment, as shown in Algorithm 3. A PWC agent allows inbound connections and already established sessions to proceed during containing its local host. Containment in PWC should handle two types of packets that indicate outbound connection attempts: outbound UDP and outbound TCP SYN packets. During the containment, a PWC agent first tries buffering the connection attempts, to forward them when the containment is relaxed. The buffered connection attempts
will be dropped with appropriate handling if the buffer becomes full or if the packets are delayed for longer than predefined timeout (up to a couple of seconds). Meanwhile, PWC needs a special handling to integrate itself seamlessly with other network-based signature identification and filtering techniques [6, 7, 8, 12]. When a PWC agent buffers a connection request, it also forwards a copy of the packet if the destination address is not in the same enterprise network. Since the copy of connection request should not be received by the destination host, the PWC agent replaces the TTL value with the number of hops to the border of network. Given the address of the border router, the agent can measure exact number of hops, using the same method as TRACEROUTE does [69]. Therefore, the signature extractor can see worm scans as if the sources were not contained, while the scan from the contained host cannot reach the victims. To prevent congestions on internal paths, the rate of forwarded copies must be controlled (Line 12).

### 3.3.4 Containment Relaxation Analysis

During the containment, a PWC agent maintains \( ddst \), the number of distinct destination addresses to which its host initiates connection attempts, to see if the host shows sustained rate higher than \( \lambda \). We call this analysis relaxation analysis since the goal is to relax mistakenly contained hosts.

#### 3.3.4.1 When to Relax Containment

Algorithm 4 depicts relaxation analysis and relaxation decision during containment. A PWC agent begins relaxation analysis at time \( t_{begin}^{contain} \) when it starts containing its local-host, and completes the analysis at time \( t_{end}^{contain} \) when it detects the first outbound connection attempt after \( t_{begin}^{contain} + t_{relax} \). The connection rate \( r_{contain} \) updated at the end of the relaxation analysis is defined as \( \frac{ddst}{|contain|} \), where \( |contain| = t_{end}^{contain} - t_{begin}^{contain} \). If \( r_{contain} \) is lower than \( \lambda \), the containment should be relaxed. Otherwise, relaxation analysis should be performed again. Consecutive \( F \) failures in relaxing containment will let the host isolated from the network. PWC is resilient to a worm that repeats scan at a burst rate and sleep [70] since relaxation analysis does not let the containment relaxed until \( r_{contain} \) is slowed down to \( \lambda \) scans per second as shown in Line 7. \( \beta \) is used to tune generosity of relaxation.
Algorithm 4 RELAXATION_ANALYSIS(pkt)

Require: pkt = outbound packet from local host, dsthist = known dst IP addresses, srchist = known src IP addresses, ddst = distinct destination count, 
\( t_{\text{contain}} \) = times when containment started, \( F \) = max relaxation failures.

if \( \text{pkt.dst.ip} \notin \text{srchist} \cup \text{dsthist} \) then
    \( \text{ddst} \leftarrow \text{ddst} + 1 \)
end if

if \( \text{pkt.time} > t_{\text{contain}} + t_{\text{relax}} \) then
    \( |\text{contain}| \leftarrow \text{pkt.time} - t_{\text{begin}} \)
    \( r_{\text{contain}} \leftarrow \frac{|\text{contain}|}{\text{ddst}} \)
    if \( r_{\text{contain}} < \beta \cdot \lambda \) then
        Relax the containment.
    else if it is \((F + 1)\)th relaxation failure then
        Isolate local host.
    else
        Continue the relaxation analysis.
    end if
end if

3.3.4.2 Effect of Incorrect Relaxation

When the scan rate of a worm is very close to the scan threshold \( \lambda \), an infected host could be relaxed mistakenly after one or more rounds of relaxation analysis. We will discuss a multi-threaded TCP-based worm that would be relaxed in one round of relaxation analysis as an worst-case example.

A local PWC agent of an infected host would start containment and relaxation analysis after detecting \( N \) scans (TCP SYN) sent by the host. Then, the PWC agent begins to buffer additional scan packets sent by the worm until all threads of the worm wait for connection responses (TCP SYN-ACK, ICMP type-3, or timeout). In the worst case, the PWC agent would relax the containment and forward all buffered scan packets in one round of relaxation analysis. Let us assume \( t_{\text{relax}} \) is less than TCP timeout value and the number of scan packets to be buffered is \( W \). Given that \( t_d \) is the detection delay for initial \( N \) scans, \( R \), the rate of the scan packets that would escape from the host is defined as,
\[ R = \frac{N + W}{td + t_{relax}} \]  

(3.1)

Since the infected host is once contained and relaxed, we have,

\[ \frac{N}{td} > \lambda \iff td < \frac{N}{\lambda} \quad (td > 0, \lambda > 0) \]  

(3.2)

\[ \frac{W}{t_{relax}} < \lambda \iff W < \lambda \cdot t_{relax} \quad (t_{relax} > 0) \]  

(3.3)

From (3.2), (3.3), and (3.1), \( R \) has its maximum value when \( td \to 0 \) and \( W \to \lambda \cdot t_{relax} \). Thus, we have the following conclusions:

\[ R < \frac{N}{t_{relax}} + \lambda \]

Therefore, under PWC, the scan rate of a worm would be suppressed below \( \frac{N}{t_{relax}} + \lambda \) scans/sec even in the worst case, and \( W \to 1 \) in the case of a single-threaded worm.

### 3.4 Experiment Setup

Symbols and notations used in following sections are described in table 3.1. Through extensive simulations on enterprise-level real traces, we have evaluated (1) cost-effectiveness of PWC; (2) effect of collaboration; and (3) impact of partial deployment. We also implemented a prototype PWC agent\(^1\) to study the impact on local P2P traffic. We have used following three metrics through out the evaluation:

- M1, the number of hosts infected by local worm scans.
- M2, the number of released worm scan packets.
- M3, total denial-of-service time per host.

We evaluate PWC against two existing techniques, Williamson’s Virus Throttle [1] and Hamsa [12] in terms of each metric. Virus Throttle generates false positives on seven hosts in the background traffic, thus we set up another configuration

\(^1\)Current prototype does not implement collaboration.
Table 3.1. Notations used in evaluation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>x</td>
</tr>
<tr>
<td>$V$</td>
<td>The number of vulnerable hosts in the enterprise network.</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Indicator threshold (unique destinations/sec)</td>
</tr>
<tr>
<td>$N$</td>
<td>The number of scan suspects to calculate $r$.</td>
</tr>
<tr>
<td>$nMI$</td>
<td>The number of mistakenly isolated uninfected hosts.</td>
</tr>
<tr>
<td>$rAC$</td>
<td>The rate of active containments at a host.</td>
</tr>
<tr>
<td>$rPC$</td>
<td>The rate of passive containments at a host.</td>
</tr>
<tr>
<td>$nI$</td>
<td>The number of infected hosts in the network.</td>
</tr>
<tr>
<td>$fI$</td>
<td>$\frac{nI}{V}$</td>
</tr>
<tr>
<td>$nI_0$</td>
<td>The number of initially infected hosts.</td>
</tr>
<tr>
<td>$nES$</td>
<td>The number of escaped scans.</td>
</tr>
<tr>
<td>$rS$</td>
<td>Worm scan rate.</td>
</tr>
<tr>
<td>$L$</td>
<td>Worm’s local preference (0.0 for uniformly scanning worms).</td>
</tr>
<tr>
<td>$rD$</td>
<td>Average delay per connection request at a host.</td>
</tr>
<tr>
<td>WIL-$x$-$y$</td>
<td>Virus Throttle with $</td>
</tr>
<tr>
<td>PWC-$x$-$y$</td>
<td>PWC with $\lambda = x$ and $N = y$.</td>
</tr>
</tbody>
</table>

WIL-$5$-$1500$ besides the default WIL-$5$-$100$. WIL-$5$-$1500$ is the most conservative configuration that does not raise false positives with the tested normal traffic. We deployed Hamsa at the border of the enterprise network in the simulator. Hamsa starts generating signatures when the suspicious pool size reaches 500 and the signature extraction takes 6 seconds [12].

- **Hypothetical Enterprise Network**: The enterprise network simulations assume /16 local address space and 13,000 hosts with $V = 6,500$. We assume no inbound scans from external infectee, for PWC is an unidirectional worm containment approach. Also, Round-Trip-Time (which is typically less than 1 ms) within the same enterprise network is ignored.

- **Background Traffic**: To configure parameters and to render normal traffic, we have used six traces of the Auckland-IV traces [66] collected on Feb 21 (BG1), Mar 7 (BG2), Mar 14 (BG3), Mar 21 (BG4), Mar 28 (BG5), and Apr 4 (BG6) in 2001. The traces collected at the border of the University of Auckland do not contain local-to-local traffic. The omitted traffic would not affect the experiment results since the observation on our own local network running PWC agent prototype showed that (1) the local-to-local inbound
and outbound connections implied high locality that could be filtered by
inconhist and outconhist; and (2) the burst rate of normal outbound con-
nection attempts did not sustain. In addition, the omitted traffic will also
affect existing techniques being compared with our system.

- **Test Worms**: Three types of test worms include (T1) randomly uniformly
scanning worms, (T2) 0.3 local preferential scanning worms, and (T3) 0.5
local preferential scanning worms. T2 and T3 worms give idea of PWC’s
effect on the local preferential scanning worms in real world. For example,
the CodeRed-II worm scans the same /8 network with 50% probability and
scans the same /16 network with 37.5% probability. The Blaster worm picks
the target within local /16 network at a probability of 40% and a random IP
at 60%.

3.5 Evaluation

3.5.1 Tuning Parameters

First, we tune indicator threshold $\lambda$, size of the outbound contact history $N$, and
relaxation analysis duration $t_{relax}$ based on normal traffic. We use BG1 for this
purpose.

3.5.1.1 The Indicator Threshold

The criterion that we used for a good $\lambda$ was the number of mistakenly isolated
hosts $nMI$ which was, in other words, the false positives that relaxation analysis
could not handle. We calculated $nMI$ varying $\lambda$ and $N$, while running PWC on 24
hours of BG1 trace. Given that $N = 5$ to reduce the effect of false alarms caused
by $N$, we observed $nMI > 3$ when $\lambda < 7$. When $\lambda \geq 7$, $nMI$ was zero even with
$N = 2$ for the most aggressive configuration.

3.5.1.2 The Size of the Outbound Contact History

Small $N$ enables rapid containment while raising more false positives. Consequent
repeated active/passive containments would increase availability-loss.
For each host, we calculated $rAC$ and $rPC$, per-minute rates of active and passive containments caused by false indicators, varying $N$, running 24 hours of BG1 trace. Since propagated false indicators could be discarded by the vulnerability window analysis, $rPC$ at each host would be the rate at which the vulnerability window analysis would yield UNSAFE on propagated false indicators. $rAC$ and $rPC$ at each host is shown in Figure 3.6. For clarity, we picked top three hosts in $rAC$, and five of the top 2% hosts in $rPC$.

In the results shown in Figure 3.6, $rAC$ and $rPC$ were stable in the range where $4 \leq N \leq 9$. $rPC$ was less than once in ten minutes at more than 98% of entire hosts when $N = 4$, and more than 99% when $N = 10$. Based on the results, we could set $N$ to 4 for a conservative configuration and 10 for a less conservative yet more accurate configuration.

### 3.5.1.3 The Relaxation Analysis Duration

The relaxation analysis duration $t_{relax}$ is closely related with the overall time in which each host would be contained.

Let us denote by $\phi_i$ the sum of $\phi_{i,j}$ at host $i$, where $\phi_{i,j}$ represents the length of the $j^{th}$ containment at host $i$. Figure 3.7 shows the spread of $\phi_i$ for various $t_{relax}$ at each uninfected host monitored during the busiest 14,400 s of BG1 trace. The result suggests a smaller $t_{relax}$ would reduce the availability-loss shown as $\phi_i$. In addition, a longer $t_{relax}$ would require a PWC agent to allocate larger
buffer to hold more outbound connection attempts during containment. As the result, more packets would be timed out and dropped within the buffer, which would increase also denial-of-service effects. However, $t_{relax}$ should not be too small because containment would be always relaxed if the worm being contained attempts less than $\lambda \cdot t_{relax}$ outbound connections during the relaxation analysis. For such a worm, PWC is still able to suppress the scanning rate under $(\frac{N}{t_{relax}} + \lambda)$ scans/sec in the worst case, as we discussed in Section 3.3.4.2. Therefore, we deliberately configure $t_{relax}$ to be 1 s, where $\phi_i$ at 99.9% of hosts are less than 32 s or 0.2% of the simulated time.

### 3.5.2 Performance Evaluation

We evaluated worm containment performance of PWC system in case of worm outbreak in the enterprise network where PWC was fully deployed. Six of Auckland-IV traces (BG1 to 6) collected on different dates are used to render every day non-worm traffic, from which we can see how PWC affects naive network traffic. WIL-5-100 isolated seven uninfected hosts that appeared in the background traffic and WIL-5-1500 introduced significant delays on them. In contrast, all the false positives raised by PWC agents were resolved during subsequent relaxation analysis within 1 s.
Figure 3.8. The number of infected hosts for local-preferential scanning worms with various scan rates. $V = 6,500, nI_0 = 10$

### 3.5.2.1 Containment Performance

To evaluate containment performance, we set up 10 initially compromised hosts in a simulated enterprise network with 13,000 hosts, and run PWC and the competitors until either all the vulnerable hosts are infected or all the worms are contained. Six background traffic traces (BG1 to BG6) are used to simulate naive network traffic in the enterprise network. For each of BG1 to 6, we repeat the simulation 100 times, and compute average of each metric. Background traffic is randomly selected within the busiest 6 hours (from 10:00am to 4:00pm) of each of the six traces.

**3.5.2.1.1 M1 (Local-to-Local Infection Rate)** The most significant contribution of PWC is the suppression of local-to-local worm infection. It could be extreme to assume $nI_0 = 10$ and 50% of entire hosts were susceptible. Nevertheless, as shown in Figure 3.8, PWC successfully suppressed local-to-local infections for T2 and T3 worms. Results of all the six background traffic traces are shown in Figure 3.9. We can also see that worm scan rate did not affect PWC’s performance.
3.5.2.1.2 M2 (Escaped Worm Scans) A successful worm containment strategy must minimize the number of scans that escapes the perimeter of defense during the delay when the containment system detects the enemy and prepares its weapon (i.e., signatures). We measured the number of escaped scans for each of PWC, Virus Throttle, and Hamsa, until the worm propagation was completely stopped. Figure 3.10 shows PWC outperformed Virus Throttle and Hamsa in terms of M2,
the number of escaped scans. While Virus Throttle and Hamsa performed better for the faster worms and the slower worms respectively, PWC showed consistent performance for all the tested scan rates. As the worm scanned local address space more aggressively, the performance gap between PWC and other techniques became more significant. WIL-5-100 seemed to perform better than PWC-7-10 for the worms whose scan rates were faster than 25 to 50 scans per second. However, WIL-5-100 isolated 7 hosts due to the false positives. We observed no naive host had been isolated by PWC during the simulations on M2. M2 computed from different background traffic is also presented in Figure 3.11.

3.5.2.2 Impact on Uninfected Hosts

We compared PWC in terms of availability-loss that the containments caused by false indicators introduced. Please note that, in spite of its longer detection delay, Hamsa does not introduce the availability-loss. Therefore, we evaluated availability-loss that PWC introduced, in comparison with Virus Throttle only.
Figure 3.11. The number of escaped scans for different background traffic (BG1 to 6). $V = 6,500, nI_0 = 10$

3.5.2.2.1 M3 (Total Denial-of-Service Time Per Host) To compare PWC and Virus Throttle, we calculated $rD$, the average delay per request at each naive host, for both systems running on each of 24-hour traces BG1 to 6. Then, we sort the hosts according to $rD$.

Figure 3.12 shows the average of six $rD$'s listed on the same rank, which shows that PWC significantly outperforms WIL-5-100 and WIL-5-1500 in terms of M3.
Due to the long delay queue, WIL-5-100 and WIL-5-1500 delayed outbound connection requests for couples or even tens of seconds in average at several hosts while the maximum $rD$ was 0.97 s/request for PWC-7-4 and 0.5 for PWC-7-10. The variations were 0.0018 and 0.0004 for PWC-7-4 and PWC-7-10 respectively. Per-request delays for 99.80% of the hosts in the PWC-7-4 experiment were less than 0.4 s, and for 99.99% in the PWC-7-10 experiment were less than 0.3 s.

### 3.5.3 Impact of Indicator Propagation Delay

To see the impact of indicator propagation delay, we studied the worst case. In particular, for each of PWC-7-4 and PWC-7-10, we set up two different PWC systems: one propagated indicators in LAN speed, while the other propagated no indicators. In each system, we infected 10 hosts with each of worms T1, T2, and T3. We performed 100 simulations in each experiment and compared the performance in terms of M1 (Figure 3.13) and M2 (Figure 3.14).

For T1, we observed no significant impact of indicator propagation delay because all the infected hosts were contained before infecting any local victims. Similar phenomena were observed in the cases of PWC-7-4 with T2 and T3. However, in the cases of PWC-7-10 with T2 and T3, indicator propagation improved performance by 4.6-19.8% and 4.3-10.6% in terms of M1 and M2 respectively. The results suggested the performance improved as worms’ preference to the local addresses increased.
Figure 3.13. Indicator propagation effect in terms of the number of infected hosts. $V = 6,500$, $nI_0 = 10$

Figure 3.14. Indicator propagation effect in terms of the number of escaped scans. $V = 6,500$, $nI_0 = 10$
Figure 3.15. The number of infected hosts for different deployment ratio (in log-scale). \( rS = 25 \) scans/sec, \( V = 6,500 \), \( nI_0 = 10 \)

### 3.5.4 Impact of Partial Deployment of PWC

The impact of partial deployment of PWC is shown in Figures 3.15 and 3.16 in terms of M1 and M2 respectively. We performed experiments on different deployment scenarios where 40%, 60%, 80% and 100% of hosts in the enterprise network were running PWC agents.

Compared with fully-deployed PWC, the performance of partially-deployed PWC would degrade in linear speed with respect to the deployment percentage.

In addition, we compared the effectiveness of partially-deployed PWC with fully-deployed Hamsa and partially-deployed Virus Throttle. Our comparison focused on the time window \( \Pi \) that starts when the worm attack is mounted and ends when Hamsa generates the signature, for PWC will (typically) terminate itself as soon as the signature for the worm is generated. In particular, we first infected 10 unprotected hosts with a worm sending 25 scans/sec. Then we defined \( \Pi \) by running Hamsa. Finally, we measured M1 and M2 of Virus Throttle and PWC during \( \Pi \).

Overall, partially-deployed PWC-7-4 performed substantially better than fully-
deployed Hamsa; and PWC-7-4 showed better or equal performance than Virus Throttle in all the partial deployment scenarios. Note that Virus Throttle’s performance could be worse since it would introduce certain delay in Hamsa’s suspicious traffic collection phase while PWC would not.

3.5.5 Impact on P2P Traffic

Behavior of a P2P file-sharing application is similar to that of a worm in that it exchanges packets with a large number of peers in sending, receiving, and searching desired data. We implemented prototype of a PWC agent, and executed it in a PC running a P2P file-sharing application to see the impact of PWC.

We tested a prototype PWC agent with eMule 0.47c, a P2P file sharing application in the eDonkey network and the (server-less) Kad network. The prototype was installed in a Winsock LSP layer of a PC running Windows XP Professional SP2 connected to Internet through a cable modem.

During 9.35 hours of the experiment, eMule was sharing 494 files (130.31MB in average), with unlimited uploading and downloading speed (effectively 60-70
KBytes/sec and 120-130 Kbytes/sec respectively). With the heuristic in Section 3.3.1.3, logarithmic increase of 3,806 distinct peers were monitored in 10,990 connection attempts. eMule released excessive UDP packets shorter than 150 bytes in the Kad network mainly to search sources and peers. Due to the performance issues, it is natural for P2P protocols to use short packets to deliver such extremely frequently transmitted messages. Our indicator detector ignored them based on the length threshold (200 bytes).

The analysis on P2P traffic suggested that P2P clients can be discriminated from worms in (a) short UDP packet size, (b) large but limited number of distinct destinations, and (c) the ratio of distinct destinations to total destinations. Our prototype leveraged (a) and (b) to eliminate P2P traffic in detecting indicators. As the result, eMule had been contained 11 times during the first 1.08 hours and never been contained afterwards. Only 0.28% of the outbound connection attempts were delayed (but not discarded) 1.35 s in average (maximum 2.09 s).

3.6 Security Analysis

3.6.1 Indicator Injection Attacks

There are two types of indicator injection attacks: a spoofed indicator injected from an external host; a forged indicator from a internal host compromised by an attacker.

3.6.1.1 Indicator Injection from an External Host

In this attack mode, we assume the attacker has no knowledge about the authentication keys used in indicator propagation. A proper firewall configuration that ensures no incoming packet has any internal source address can filter simple injection of bogus indicators spoofing internal source address. However, a sophisticated external attacker may use IP tunnel. Thereby, PWC uses a public-key authentication scheme to authenticate each PWC agent to the PWC manager and vice versa. The scheme is scalable in that (1) PWC agents only need to verify the validity of the PWC manager’s certificate and (2) the PWC manager knows whether or not an agents certificate has been revoked as the PWC manager issues certificates to
PWC agents. Although public-key authentication operations may consume CPU cycles, fast RSA [71] in current hardware can authenticate within 100 µsec and enterprise hosts typically have enough CPU power. As shown in Figure 9 and Figure 10, the impact of small delay on PWC is negligible. (Note that symmetric-key authentication is not as much manageable regarding the number of PWC agents.)

3.6.1.2 Indicator Injection from an Internal Host

In this attack mode, we not only upgrade the attacker’s capability so that he may “steal” the key from a compromised host, but also assume insider threat. Under a public-key authentication scheme, the attacker can forge an indicator either by invoking the signing subroutine of the (local) PWC agent, or by “stealing” the private key used by the compromised host.

3.6.1.2.1 The First Attempt As we will review shortly in Section 3.6.3, how a PWC agent is implemented determines the ability of an internal attacker to “fool” the PWC agent. Because a PWC agent accepts and processes only restricted part of packet headers, the agent program itself is less likely to have buffer related vulnerabilities. When implemented in software, modern computer architecture and operating systems practices also provide a variety of memory protection measures that should be applied to protect PWC agents. By taking these protective measures, the first attempt would have a high probability in crashing the system instead of “fooling” the system. More conservatively, we could implement a PWC agent as a piece of firmware inside a NIC card or a smart switch.

3.6.1.2.2 The Second Attempt requires a lot of knowledge on where and how the private key is stored. This threat leads us to the fundamental problem of runtime key protection that is well studied in the literature. If the private key has to be stored in the memory, we have two defenses: use of key obfuscation techniques to store keys encoded, or use of key partitioning to break the key into several parts and store them in different memory locations. Another way is to store the private key using a tamper resistant hardware module such as Trusted Platform Module in the TCPA architecture [72]. The private key and hashed values of the trusted entities are sealed within the module. As Garfinkel et al. discussed in
[73], the module will release the private key only to the trusted entities with the same hashes that sealed it. The key will be safe if the attackers do not know how to use the module.

### 3.6.2 False Positive Indicators

Three possible reasons may cause a PWC agent to raise false positive indicators: a compromised agent that is already addressed in the previous section; a burst rate in connection attempts that is already handled by indicator detector and relaxation analysis; special applications such as proxy servers – note that P2P/VoIP/Audio/Video streaming clients and Instant Messengers raise few or no false indicators by Algorithm 1.

Proxy servers that connect external clients with internal servers scarcely detect false indicators once internal servers start occupying $dsthist$. However, proxy servers serving internal clients often connect to various external servers and may cause false indicators. In this case, instead of disabling PWC agents at the proxy servers, we can apply a proper firewall configuration to deny any requests from external clients, thus preventing external worms. As shown in Figure 3.8, internal worms hardly reach internal hosts including proxy servers before being contained.

Finally, for applications/services that seldom generate bursts in connection requests, those seldom experienced extra delay will be minimal as in Figure 3.12.
3.6.3 Worms May Bypass/Disable the Agent after Compromising the Host

Let us assume that we implement a PWC agent in one of the Layers in Figure 3.17. A sophisticate worm may try following attempts to neutralize local PWC agent after they successfully break into a host: attempts to bypass; or disable.

3.6.3.1 Bypass Attempts

Security measures implemented in Layers 1 and 2 are vulnerable to the bypass attempts of a worm that can directly access Layer 3 or Layer 4 interfaces. However, it is not feasible for a worm to bypass Layer 3 since, in order to spread among asymmetric systems, the worm must be able to access variety of NICs directly without using drivers. Bypassing Layer 4 is also impossible since the worm must use NIC. Thus, if the PWC agent is implemented in either one of Layers 3 and 4, it should be very difficult or even impossible for the worms to bypass the agent.

3.6.3.2 Disable Attempts

When a worm exploits one of kernel-mode vulnerabilities, it could have power to (a) unload any programs including drivers; and (b) access full address space. Power (a) allows the attackers to write a worm that propagates freely after unloading PWC agents. However, if the drivers that are necessary for propagation are unloaded, the worm will be contained. Thus, PWC agents can be embedded in those necessary drivers such as Layer 3. Note that there are as many NIC drivers as the number of NIC products. Assuming that the vendors embed PWC agents in their drivers, it could be almost impossible for a worm to try power (b) to disable the agent without crashing the driver: although the worm might disable a few drivers, it becomes a partial deployment scenario at most.

A better way to prevent a worm from bypassing or disabling PWC is to implement PWC agent off-the-box. A PWC agent can be embedded in firmware of NICs (Layer 4 in Figure 3.17), or in subnet routers. It might be challenging to implement such functionalities in a extremely loaded devices (such as Gbps or tens of Gbps links), which might make it difficult to fully deploy PWC in an enterprise network. However, we have shown PWC is still better than existing techniques un-
der partial deployment scenarios in Section 3.5.4. In addition, today’s sophisticate switches are flexible and powerful enough to run software components.

3.6.4 Other Counterattacks

3.6.4.1 Poisoning Attacks

Attackers may try *UDP-flooding attack* in which an internal or external attacker sends excessive UDP packets to a protected host to keep the recipient’s vulnerability window analysis yielding UNSAFE. Then, any propagated indicator received at the host will lead to passive containment. However, this attack cannot be successful since PWC controls the frequency of propagated indicators. Moreover, attackers cannot mimic propagated indicators as in Indicator Injection Attacks.

3.6.4.2 Replacement Attacks

Replacement attacks are to overwrite (or erase) PWC agents stored in file systems. Worms must use system-calls or BIOS service routines to access file systems. So, we may have the system-calls deny attempts to overwrite PWC agents. BIOS also can help by storing the PWC agent in a restricted area on a disk and denying any unauthorized write-access to the area. In addition, *Microsoft Windows Vista* and *Linux* support Trusted Platform Module to protect file systems from unauthorized changes.

3.6.4.3 Wait-before-Scan Attack

A worm may try waiting a prolonged period before starting scanning to evade passive containment. This attempt cannot let the worm successfully spread out because the PWC agent at the host will initiate active containment after the worm releases the first $N$ scans.
3.7 Discussions

3.7.1 Applicability

Besides uniformly scanning worms, PWC can successfully suppress topologically aware worms, hit-list worms, flash worms, polymorphic worms, metamorphic worms, etc. that scan more than $\lambda$ new addresses per second. PWC agents are light-weight so that they can be implemented in either way of hardware or software component. Other worm defense measures can be run in parallel with PWC since PWC agents can still forward large part of malicious messages during containment. PWC guarantees those forwarded malicious messages will not infect any host. Finally, PWC allows P2P traffic.

3.7.2 Limitations

PWC also has several limitations. First, as a host-based approach, PWC requires a majority of internal hosts to run PWC agents. However, the performance degradation in various partial deployment scenarios is not worse than existing techniques. Second, proxy servers of specific type need to be protected in an alternative way as mentioned in Section 3.6.2. Third, as a certificate authority, the PWC manager must be running in a highly secured host. However, without the PWC manager, the performance of PWC is still acceptable as we discussed in Section 3.5.3. Finally, during the containment, a PWC agent may experience stalled-scan problem in which a worm’s scanning rate is slowed down. This could let the PWC agent relax an infected host after performing a couple of rounds of relaxation analysis. However, this is a problem limited only to the TCP-based worms scanning in a synchronous manner, and PWC can still slow down those worms.

3.8 Conclusion

In this study, we proposed PWC, a proactive worm containment solution for enterprises. With aggressive containment and subsequent relaxation analysis based on two novel white-host detection techniques, PWC could stop an infected host after merely 4 to 10 scans were released while minimizing collateral denial-of-service
effects. Evaluation based on real traces and extensive worm simulations demonstrated that PWC significantly outperformed Virus Throttle [1] in terms of all of three metrics and Hamsa [12] in terms of local-to-local infections and local-to-remote infections. In partial deployment experiments, PWC outperformed Virus Throttle.
Chapter 4

Value Based Software Plagiarism Detection

4.1 Introduction

Identifying same or similar code fragments among different programs or in the same program is very important in some applications. For example, duplicated codes found in the same program may degrade efficiency in both development phase (e.g., they can confuse programmers and lead to potential errors) and execution phase (e.g., duplicated code can degrade cache performance). In this case, code identification techniques such as clone detection \cite{41, 42, 43, 44, 45, 46, 47, 48} can be used to discover and refactor the identical code fragments to improve the program. For another example, same or similar code found in different programs may lead us to even more serious issues. If those programs have been individually developed by different programmers, and if they do not embed any public domain code in common, duplicated code can be an indication of software plagiarism or code theft. In code theft cases, determining the sameness of two code fragments becomes much more difficult since plagiarizers can use various code transformation techniques including code obfuscation techniques \cite{36, 37, 38} to hide stolen code from detection. In order to handle such cases, code characterization and identification techniques must be able to detect the identical code (i.e., two code fragments belonging to the same lineage) without being easily circumvented by code transformation techniques.
Previous works are largely insufficient in meeting all of the following three highly desired requirements: (R1) Resiliency to automated semantics-preserving obfuscation tools [39, 74, 75, 40]; (R2) Ability to directly work on binary executables of suspected programs. In some applications such as code theft cases, source code of plaintiff’s program is usually available, but the source code of suspect software products often cannot be obtained until some strong evidences are collected; (R3) Platform independence, e.g., independent from operating systems and programming languages. As we can see in the related work section, the existing schemes can be broken down into four classes to see their limitations with respect to the aforementioned three requirements: (C1) static source code comparison methods [56, 58, 50, 51, 52, 53, 54, 55]; (C2) static executable code comparison methods [57]; (C3) dynamic control flow based methods [59]; (C4) dynamic API based methods [62, 60, 61]. First, Class C1, C2 and C3 do not satisfy requirement R1 because they are vulnerable to semantics-preserving obfuscation techniques such as outlining and ordering transformation. Second, C1 does not meet R2 because it has to access source code. Third, the existing C3 and C4 schemes do not satisfy R3 because they rely on features of Windows or Java.

To address the above issues, we introduce a novel approach to dynamic characterization of executable programs. After we examined various runtime properties of executable programs, we found an interesting observation that some runtime values of a program are hard to be replaced or eliminated by semantics-preserving transformation techniques such as optimization techniques, obfuscation techniques, different compilers, etc. We call such values core-values.

To investigate the resilience of core values (to semantics-preserving code transformation), we generated \(e_{1..5}\), five different versions of executable files of test program \(p\) written in C, by compiling \(p\) with each of the five optimization switches of GCC (\(-00\), \(-01\), \(-02\), \(-03\), and \(-0s\)). From each of \(e_{1..5}\) given the same test input, we extracted a *value sequence*, a sequence of values (4-bit, 8-bit, 16-bit, or 32-bit) written as computation results of arithmetic instructions and bit-wise instructions in the execution path. As a way of retaining (in the value sequence) only the values derived from input, we implemented a dynamic taint analyzer.\(^1\) When we

\(^1\) We also have noticed that there are studies on identifying and overcoming limitations of dynamic taint analysis. Please note that dealing with those limitations is out of our scope.
analyzed the value sequences of $e_{1..5}$, we found that some values survived all of the five optimization switches. Moreover, the sequence of the values surviving all of the five optimization switches was enclosed almost perfectly by the value sequences of executables generated by compiling $p$ with different compilers (we tested Tiny C Compiler [76] and Open Watcom C Compiler [77]). This indicates that core-values do exist and we can use them to check whether two code fragments belong to the same lineage.

In this study, we show (1) how we extract the values revealing core-values; and (2) how we can apply this runtime property to solve problems in software plagiarism detection. We implemented a value extractor with a specific dynamic taint analyzer and value refinement techniques atop a generic processor emulator, as part of our value-based program characterization method. As a machine code analyzer that directly works on binary executables, our technique satisfies R2. Because our technique analyzes generic characteristics of machine instructions, it satisfies R3. Regarding R1, we implemented a value-based software plagiarism detection method (VaPD) that uses similarity measuring algorithms based on sequences and dependence graphs constructed from the extracted values. We evaluated it through a set of real world obfuscators including two commercial products, Zelix Pty Ltd.’s KlassMaster [40] and Semantic Designs Inc.’s Thicket [75]. Our experimental results indicate that the VaPD successfully discriminated 34 plagiarisms obfuscated by SandMark [39] (totally 39 obfuscators, but 5 of them failed to obfuscate our test programs); plagiarisms heavily obfuscated by KlassMaster, C programs obfuscated by the Thicket C obfuscator, and executable files obfuscated by the control flow flattening implemented in the Loco/Diablo link-time optimizer [74].

**Contributions:** (1) We present a novel code characterization method based on runtime values. To our best knowledge, our work is the first one exploring the existence of the core-values. (2) By exploiting runtime values that can hardly be changed or replaced, our code characterization technique is resilient to various control and data obfuscation techniques. (3) Our plagiarism detection method (VaPD) does not require access to source code of suspicious programs, thus it could greatly reduce plaintiff’s risks through providing strong evidences before

---

2Since SandMark and KlassMaster work on Java bytecode, we use GCJ, GNU ahead-of-time compiler for Java, to convert obfuscated programs to x86 native executables.
Table 4.1. Proportion of refined value sequences of GCC compiled executables that overlap value sequences of TCC and WCC compiled executables.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Optimization switches tested</th>
<th>bzip2</th>
<th>gzip</th>
<th>oggenc</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCC</td>
<td>NA</td>
<td>100%</td>
<td>100%</td>
<td>92%</td>
</tr>
<tr>
<td>WCC</td>
<td>20 switches</td>
<td>100%</td>
<td>100%</td>
<td>&gt; 91% (avg. 95%)</td>
</tr>
</tbody>
</table>

filing a lawsuit related to intellectual property.

This chapter is organized as follows. In the next section, we briefly discuss related works. In Section 4.2, we discuss the existence of core-values implied by our preliminary experimental results. In Section 4.3 and 4.5, we evaluate our method by applying it to the problems of software plagiarism detection. Finally, the limitations, some potential counterattacks, and future work are discussed in Section 4.6.

4.2 Do Core-Values Exist?

The runtime values of a program are defined as values from the output operands of the machine instructions executed. While examining the runtime values of executable programs, we established the following hypothesis of a synthesized runtime characteristic that can be viewed as a dynamic birthmark of code fragments.

**Hypothesis 1.** There exist some runtime values of a program that cannot be changed through automated semantics-preserving transformation techniques such as optimization, obfuscation, different compilers, etc.

We call the values satisfying the above hypothesis core-values. Core-values of a program are constructed from runtime values that are pivotal for the program to transform its input to desired output. Although it is an open problem to formalize and prove the concept of core-values, we can practically eliminate non-core values from the runtime values to retain core-values. To identify non-core values, we leverage taint analysis and easily accessible semantics-preserving transformation techniques such as optimization techniques implemented in compilers. Let $v_P$ be a runtime value of program $P$ taking $I$ as input, and $f$ be a semantics-preserving transformation. Then, the non-core values have the following properties:

(i) If $v_P$ is not derived from $I$, $v_P$ is not a core-value of $P$. 
(ii) If \( v_P \) is not in the set of runtime values of \( f(P) \), \( v_P \) is not a core-value of \( P \).

To test our hypothesis on the existence of core-values, we perform a dynamic analysis on three test programs gzip, bzip2, and oggenc: Gzip and bzip2 are well-known compression utilities, and oggenc is a OggVorbis audio format encoder. For the dataset to be used as the input to the programs, we generate ten wav audio files (seven 16KB files, two 24KB files, and one 8KB file), cropped from a 43.5MB wav file containing an 8’37”-long speech. In each set of experiments, we use these ten inputs, and take the average outcome as the final result. With each of the three programs, we generate five different versions of executable files by compiling it with each of the following optimization switches of GCC: -O0, -O1, -O2, -O3, and -Os.

From each of the executables given the same input, we extract a value sequence, a sequence of values (4-bit, 8-bit, 16-bit, or 32-bit) that are the computation results of arithmetic and bit-wise instructions in the execution path. We also implement refinement techniques (Section 4.3.1 and 4.3.2) including a dynamic taint analyzer to retain only the values derived from input in the sequence. Then, we refine the value sequences by computing their longest common subsequence, which contains the runtime values that survive all of the five optimization switches.

To verify that the refined value sequences are not from compiler-specific common routines such as standard C library or C startup code, we compare the refined value sequences against value sequences extracted from the same programs compiled by different compilers, Tiny C Compiler (TCC) and Open Watcom C Compiler (WCC). Compared to GCC, TCC uses different compiler components such as parser and optimizer, and support library (libtcc.a), however the code it produces borrows GCC’s runtime libraries (libc.so). WCC is a self-contained development suite implementing its own C libraries. Therefore, the code it produces does not need to use GCC’s runtime libraries. Also, WCC provides plenty of optimization options, and we test all the 20 optimization switches to examine the refined value sequences. As shown in Table 4.1, the longest common subsequence of the five sequences are enclosed almost completely by the value sequences of executables generated by compiling the same test program with TCC and WCC. Although 92% and 95% matches shown in the cases of oggenc indicate that the refined value sequences still contain some non-core values, these are much higher scores than those between irrelevant programs: as we will show shortly, the scores between
Table 4.2. Proportion of refined value sequences that overlap value sequences of executables obfuscated by Thicket and control flow flattening

<table>
<thead>
<tr>
<th>Obfuscator</th>
<th>bzip2</th>
<th>gzip</th>
<th>oggenc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thicket C Obfuscator</td>
<td>100%</td>
<td>100%</td>
<td>95%</td>
</tr>
<tr>
<td>Control Flow Flattening</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

We further investigate the core-values through real obfuscation tools. For a source code obfuscation tool, we use Semantic Designs, Inc.’s Thicket C obfuscator that implements abstract syntax tree (AST) based code transformation. Its features include, but not limited to, identifier scrambling, format scrambling, loop rewriting, and if-then-else rewriting. As a more advanced obfuscation technique, we use control flow flattening [38] implemented in Loco based on Diablo link-time optimizer [74]. Control flow flattening can transform a program as shown in Figure 4.1, in which the transformed control flow graph is hugely different from the original. As shown in Table 4.2, again our refined value sequences are almost completely enclosed by the value sequences of obfuscated executables.

To see overlapping portion of value sequences of different programs, we compare the refined value sequences of bzip2, gzip, and oggenc against irrelevant pairs (i.e., the refined value sequence of bzip2 to value sequence of oggenc optimized with -O1). In 30 comparison cases (three test programs, each of which has two irrelevant peers, five optimization switches), the value sequences of each program contain only 0% to 11% of the refined value sequences of different programs. This indicates that irrelevant programs range from 0% to 11% in our experiments.

Figure 4.1. Control Flow Flattening Example
the core-values do exist and we can use them to identify the sameness of codes.

4.3 Software Plagiarism Detection

With the rapid development of software industry and the burst of open source projects (e.g., SourceForge.net is hosting over 230,000 open source projects as of Feb. 2009), software theft has become a very serious concern to software companies and open source communities. In the presence of automated semantics-preserving code transformation tools [40, 74, 39, 75], the existing code characterization techniques may face an impediment to finding sameness of plagiarized code and the original. In this section, we discuss in detail how we apply our technique to software plagiarism detection. We evaluate our value-based code characterization method against such code obfuscation tools in the context of software plagiarism detection.

Scope of Our Work: We consider two types of software plagiarisms in the presence of automated obfuscators: whole-program plagiarism, where the plagiarizer copies the whole or majority of the plaintiff program and wraps it in a modified interface, and core-part plagiarism, where the plagiarizer copies only a part such as a module or an engine of the plaintiff program. Our main purpose of VaPD is to develop a practical solution to real-world problems of the whole-program software plagiarism detection, in which no source code of the suspect program is available and various obfuscation techniques have been applied. VaPD can also be a useful tool to solve many partial plagiarism cases where the plaintiff can provide the information about which part of his program is likely to be plagiarized. We present applicability of our technique to core-part plagiarism detection in the discussion section. We note that if the plagiarized code is very small or functionally trivial, VaPD would not be an appropriate tool.

4.3.1 Value Sequence Extraction

Since not all values associated with the execution of a program are core-values, it is important to limit the types of values to be included in a value sequence. We establish the following requirements for a value to be added into a value sequence: The value should be output of a value-updating instruction and be closely
related to program’s semantics. In the following, we discuss the rationale for these requirements.

Informally, a computer is a state machine that makes state transition based on input and a sequence of machine instructions. After every single execution of a machine instruction, the state is updated with the outcome of the instruction. Because the sequence of state updates reflects how the program computes, the sequence of state-updating values is closely related to the program’s semantics. As such, in value-based characterization, we are interested only in the state transitions made by value-updating instructions. More formally, we can conceptualize the state-update as the change of data stored in devices such as RAM and registers after each instruction is performed, and we call the changed data a state-updating value. We further define a value-updating instruction as a machine instruction that does not always preserve input in its output. For example, add is a value-updating instruction, but mov is not. Being an output of a value-updating instruction is a sufficient condition to be a state-updating value. Therefore, we exclude output values of non-value-updating instructions from a value sequence. In our x86 implementation, the value-updating instructions are the standard mathematical operations (add, sub, etc.), the logical operators (and, or, etc.), bitshift arithmetic and logical (shl, shr, etc.), and rotate operations (ror, rcl, etc.).

The above technique helps dramatically reduce the size of a value sequence; however, in practice it is still challenging to analyze all values produced by all the value-updating instructions. Therefore, we must apply further restrictions to refine value sequences. There are two classes of values computed by value-updating instructions: Class-1 includes those derived from input of the program, and Class-2 consists of those that are not. For example, when program P is processing input I in environment E, some instructions take values derived from input I as their input, but some others take input from environment E such as program load location, stack pointer, size of stack frame, etc. Since the semantics is a formal representation of the way that a program processes the input, it is obvious that the values in Class-1 are more closely related with the semantics of a program. So, we include only the values of Class-1 in a value sequence. To identify the values included in Class-1, we run a program in a virtual machine environment and perform a dynamic taint analysis [17]. We start with tainting the input, and
then our analyzer in the virtual machine propagates the taint to every byte in registers, memory cells, and files derived from the input. Registers and memory cells appearing in destination operands of all the instructions that take input from tainted registers or tainted memory locations are also tainted, and the output values of value-updating instructions are appended into the value sequence. In the example of JLex used as a case study in Section 4.5, the value sequences contain less than 7,000 values after applying taint analysis, which is significantly shorter, approximately $\frac{1}{250}$ of the original sequences.

### 4.3.2 Value Sequence Refinement

In this section, we discuss heuristics to refine value sequences. An initial value sequence constructed through the dynamic taint analysis may still contain a number of non-core values produced by intermediate or insubstantial computational steps. We need to eliminate those values to make the value sequence (1) as close to core-values as possible; and (2) capable of characterizing larger programs. We believe a number of heuristics such as control/data flow dependence analysis and abnormal code pattern detection can be adopted to achieve these goals, and below we introduce some of them. One principle that we consider here is that we have to be conservative in processing value sequences of suspect programs. Since some heuristics may be abused by sophisticated plagiarizers, we summarize applicability of each heuristic that we will introduce in Table 4.3.

#### 4.3.2.1 Sequential Refinement

Inside the value sequence extractor, we implement a refinement technique named *sequential refinement*. Figure 4.2 shows how GCC compiles “`a=1; a=(a+1)*11;`.” When variable `a` is initially tainted, our taint analysis extracts value sequence $s = \{4, 5, 10, 11, 22\}$. Note that sequence $s_{1:4} = \{4, 5, 10, 11\}$, a subsequence of $s$. 

<table>
<thead>
<tr>
<th>Refinement technique</th>
<th>Plaintiff program</th>
<th>Suspect program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential refinement</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Optimization-based refinement</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Address removal</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

| Table 4.3. Applicability of value sequence refinement techniques. | Refinement technique | Plaintiff program | Suspect program |
|---------------------------------------------------------------------|----------------------|-------------------|
| Sequential refinement                                                | √                     |                   |
| Optimization-based refinement                                        | √                     |                   |
| Address removal                                                      | √                     | √                 |
is generated by intermediate steps computing ‘(a + 1) × 11’. All the values in $s_{1:4}$ are overwritten in register eax without affecting any other memory locations until line 005. Since instructions after line 005 would never read (or be affected by) the values in $s_{1:4}$, we can remove $s_{1:4}$ from $s$ and retain only $\{22\}$. We formalize this heuristic in the following rule:

Sequential Reduction Rule: Let $i_{m,n}$ denote $m$-th instruction updating variable (register or memory) $n$. Then, we can skip logging output of $i_{m,n}$ if $n$ is never read within range $(i_{m,n}, i_{m+1,n})$. Repeat the same process until the first instruction that reads $n$ and updates a variable ($\neq n$) is executed.

Through out our experiments presented in this study, average reduction rate by the sequential refinement is 16%, and the maximum is 34%. Note that the sequential refinement only applies to plaintiff programs because, in obfuscated programs, original values could appear as the results of the intermediate computational steps.

### 4.3.2.2 Optimization-Based Refinement

Only for plaintiff programs, we perform optimization-based refinement as shown in Figure 4.3. One of the easiest way to obtain different executable files that are semantically identical is to compile the same source code with the same compiler with different optimization switches enabled. Motivated by this idea, we use sev-
eral optimized executables of the same program to sift non-core values out. With GCC and its five selected optimization flags (-00, -01, -02, -03, and -0s), we can extract five *optimized value sequences* from the plaintiff program. Each optimized value sequence has been processed with the sequential refinement while it is extracted. Then, we compute a longest common subsequence of all the optimized value sequences to retain only the common values in the resulting value sequence. As we do not assume we have access to the source code of suspect programs, this refinement heuristic is only applicable to plaintiff programs.

### 4.3.2.3 Address Removal

Memory addresses or pointer values stored in registers or memory locations are transient. For example, some binary transformation techniques such as word alignment and local variable reordering can change pointers to local variables or offsets in stack; and heap pointers may not be the same next time the program is executed even with the same input. Therefore, we do not include pointer values in a refined value sequence.

In our VaPD prototype, we implement a range checking based heuristic to detect addresses. Our test bed dynamically monitors the changes of memory pages allocated to the program being analyzed, and it maintains a list of ranges of all the allocated pages with write permission enabled. If a runtime value is found to be within the ranges in the list, VaPD discards the value, regarding the value as an address. Although this heuristic may also delete some non-pointer values, it can remove pointers to stack and to heap with no exception. Address removal heuristic is applicable to both plaintiff and suspect programs.

### 4.3.3 Similarity Metric

In the literature, there are many metrics for measuring the degree of similarity of two sequences. In our prototype, we define it based on the longest common subsequence (LCS). It should be noted that the definition of the LCS does not require every subsequence to be a continuous segment of the mother sequence. For example, both \{1, 6, 120\} and \{2, 24\} are valid subsequences of value sequence \{1, 2, 6, 24, 120\}. Let \(|\text{LCS}(s_1, s_2)|\) denote the length of the LCS of sequence \(s_1\)
Figure 4.4. Plagiarism detection process

and $s_2$. Given $v_P$, a fully refined value sequence of a plaintiff program and $v_S$, a value sequence of a suspect program, similarity score of the suspect program over the plaintiff program is intuitively defined as:

$$\text{Sim}(v_P, v_S) = \frac{|\text{LCS}(v_P, v_S)|}{|v_P|}$$

4.3.4 Design Overview

Figure 4.4 shows overall design of VaPD. Here, provided with executable files of plaintiff program $P$ and suspect program $S$, and common test input $I$, Value Sequence Extractor (VSE) extracts $v_P$ and $v_S$, the value sequences of $P$ and $S$. After refining $v_P$ and $v_S$, Similarity Detector computes Sim$(v_P, v_S)$, the similarity score of $v_P$ and $v_S$. VaPD repeats this process with different inputs (say, 10 or 20 inputs), and claims plagiarism if the average of the scores shows a significant similarity.

By default, VaPD uses value sequences $v_P$ and $v_S$ extracted through the entire execution of $P$ and $S$ respectively. However, when it deals with the cases where only part of $P$ is reused in $S$, VaPD can extract partial value sequence from only the suspicious part of $P$. To extract partial value sequences, we insert special system calls into the source code of $P$ (note that we do not assume access to the source code of $S$) to notify VSE when to start (or resume) and when to stop (or pause) extracting the value sequence. Provided by the plaintiff with the intelligence about
which part of his program is likely to be plagiarized, we can annotate plaintiff’s source code and capture the sequence from the part that is believed to be stolen.

VSE is a virtual machine that executes given program instruction by instruction. We implement two operation modes in VSE: normal mode and partial extraction mode. In the normal mode, VSE operates as follows. After fetching an instruction, Taint Analyzer taints the destination operands if any of the source operands is tainted. After the instruction is executed by the virtual machine, VSE checks whether the instruction is a value-updating instruction and whether its output is tainted; if this is true, the output of the instruction is added to the value sequence. VSE then fetches and decodes the next instruction and repeats the same process until the program is finished. When the program terminates, VSE stops extracting values and passes completed value sequence to VaPD. Note that VSE also performs the address removal refinement. In the partial extraction mode, VSE intercepts two special system calls \texttt{START\_EXTRACT()} and \texttt{STOP\_EXTRACT()} (system call numbers are 0xFFFFFFFF0 and 0xFFFFFFFF0 respectively) requested by the test program. When VSE starts in the partial extraction mode, value sequence recording is initially turned off. It starts (or resumes) recording values when the test program requests \texttt{START\_EXTRACT()} system call, and it stops (or pauses) storing values when the program calls \texttt{STOP\_EXTRACT()} system call. Using the partial extraction mode, we can extract value sequences from part of plaintiff programs. Note that the partial extraction mode is to extract partial value sequence of plaintiff programs. Malicious plagiarizers will not be able to prevent this mode from excluding plagiarized part in value sequence extraction process.

To reduce the number of values added into the value sequence, VSE does not extract values from dynamic linked libraries or shared libraries by default. However, if necessary, we can enable VSE to include specific shared libraries in the value sequence extraction because the virtual machine knows which libraries are loaded and where they are.

### 4.3.5 Addressing Reordering Attacks

As we will show shortly in Section 4.5, our preliminary results show that the runtime value sequence based method is resilient to various obfuscation techniques.
However, our metric LCS is sensitive to reordering. For example, an adversary can reduce the length of common sequences by exchanging the order of independent instructions or independent basic blocks. To defend this, we propose a technique to organize a value sequence into subsequences showing unchangeable partial ordering of the values. To get such subsequences that cannot be reordered, we build specific value dependence graphs (dynamic data dependence graphs of the runtime values) denoted by VDG. Then we use a novel path containment test technique to check whether the reordering-intolerant subsequences of the plaintiff program are contained in the VDG of the suspect program. Such path containment checks (and the checking results) may replace LCS metrics in measuring the similarity of two programs. Through a Lemma, we show that such containment checks are “immune” to reordering attacks.

**Definition 1 (Value Dependence Graph).** Given a program $P$, its value dependence graph $VDG(p)$ is a directed acyclic graph $G(V_p,E_p)$, where $V_p$ is a set of vertices each of which represents a runtime value that is an output of some instruction of $p$, $E_p$ is a set of edges $(a,b)$ such that $a \in V_p$, $b \in V_p$, $a \neq b$, and the runtime value represented by $b$ is derived from the runtime value represented by $a$.

Overall process of VDG based plagiarism detection is shown in Figure 4.5. To avoid using expensive graph matching algorithms, we develop two techniques, graph reduction and path containment test.
4.3.5.1 Path Containment Test

Given that we have plaintiff program $P$ and suspicious program $S$, we construct the value dependence graph from the runtime values (refined to expose the core-values) of each program. Because the direct cyclic edges $(v, v)$ are useless, both $VDG(P)$ and $VDG(S)$ can be made a directed acyclic graph (DAG) by removing such cyclic edges. Then, with $VDG(P)$, we chop the graph into paths starting from each leaf node to the root node. Instead of enumerating all the paths of $VDG(P)$, we pick one dynamic data dependence path per each group of paths that transform an initial input to a final output because our focus is to capture the dependence among critical values monitored when an input is processed to an output. All the values contained in a path of $VDG$ have partial ordering dependence, therefore reordering techniques cannot change their order. This turns out to be a critical observation that has deeper implications that we formalize as follows in the form of a Lemma. This lemma suggests that path containment tests can be done to get a clear win in fighting with reordering attacks.

**Lemma 1** (Path Containment). *Given two programs $P$ and $S$, if the following conditions hold, then for any path $C_P$ in the $VDG(P)$, there always exists a path $C_S$ in the $VDG(S)$ such that all the values contained in $C_P$ will be a continuous or non-continuous subsequence of $C_S$.

(i) $S$ is obtained from $P$ through semantics-preserving transformations.

(ii) The values in $C_P$ are core-values.

Overall process of the path-containment test based on Lemma 1 is shown in Algorithm 5 and Algorithm 6.

4.3.5.2 Path Selection

Instead of enumerating all the paths of $VDG(P)$, we pick one dynamic data dependence path per each group of paths that transform an initial input to a final output since our focus is to capture the dependence among critical values monitored when an input is processed to an output. This simplified comparison (Algorithm 7) could increase the chances of false positives, but the impact would be insignificant since
Algorithm 5 MATCH_PATH(p, g, i, n)

Require: Source path p, target graph g, i= current position in p, n= node to start path containment test with.
Ensure: The number of matched values.
\[
\text{if } n \text{ matches } p[i] \text{ then}
\]
\[
\text{if } i \text{ is the end of } p \text{ then}
\]
\[
\text{set } \text{stop}_\text{flag}
\]
\[
\text{return } 1
\]
\[
\text{end if}
\]
\[
num_match \leftarrow \text{MATCH_SUB}(p, g, i + 1, n) + 1
\]
\[
\text{else}
\]
\[
num_match \leftarrow \text{MATCH_SUB}(p, g, i, n)
\]
\[
\text{end if}
\]
\[
\text{return } num_match
\]

Algorithm 6 MATCH_SUB(p, g, i, n)

Require: Source path p, target graph g, i= current position in p, n= node to start path containment test with.
Ensure: The number of matched values.
\[
\text{for all } c \in \text{children of } n \text{ do}
\]
\[
\quad m \leftarrow \text{MATCH_PATH}(p, g, i, c)
\]
\[
\quad num_match \leftarrow \text{maximum of } m
\]
\[
\text{if } \text{stop}_\text{flag} \text{ is set then}
\]
\[
\text{exit the loop}
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{return } num_match
\]

we perform the path containment test on hundreds of paths containing thousands of values in each. Let us denote by p the path being constructed. For each leaf node of VDG(P), we mark the leaf node as visited, and add it to p. Then we pick an unvisited predecessor of the leaf node, mark it as visited, and add it to p. If all predecessors are visited, we randomly pick one. After repeating this procedure until the root of VDG is reached, p is ready to be added to the set of selected paths.
Algorithm 7 SELECT_PATH(g)

Require: Directed acyclic graph g.
Ensure: List of selected paths.
Mark all nodes in g as not visited.
for all \( v \in \text{leaf nodes of } g \) do
  \( w \leftarrow v \)
  while \( w \) do
    Add \( w \) at the beginning of path
    for all \( p \in \text{the list of parent nodes of } w \) do
      if \( p \) is not visited then
        Mark \( p \) as visited.
        \( w \leftarrow p \)
        Exit the for loop.
      end if
    end for
    if all \( p \) were visited then
      \( w \leftarrow \text{randomly pick a parent of } w \)
    end if
  end while
  Add path to path_list.
end for
return path_list

4.3.5.3 VDG Reduction

Moreover, we further improve path matching performance through removing use-
less nodes and edges from VDG(\( S \)). When constructing VDG(\( S \)) from the runtime
values of \( S \), we remove the nodes whose values do not appear in VDG(\( P \)) by merg-
ing them to the nearest predecessors. Construction of VDG(\( S \)) is done within
\( O(E_S) \) time where \( E_S \) is the number of edges in VDG(\( S \)) since, by nature of dy-
namic trace, the nodes appearing in the dynamic dependence edges are created in
a topologically sorted order. Given the hash table containing all the values from
VDG(\( P \)), this reduction can be done on-the-fly during the construction with no
extra cost. Since the time complexity of the DFS search on a DAG is known to
be \( O(N^2) \), VDG reduction dramatically improves the matching performance. The
impact of VDG reduction is shown in the experimental results in Section 4.5.4.
4.3.5.4 VDG similarity metric

When \( p \), a path of VDG(\( P \)) is compared to VDG(\( S \)), the per-path containment score is computed as follows:

\[
P_{\text{CS}}(p, \text{VDG}(S)) = \frac{\# \text{ of matching nodes in } p}{|p|}
\]

Given a set of paths extracted from VDG(\( P \)), we use the weighted average of the per-path containment scores because the longer paths of \( P \) are more likely to serve the main purpose of \( P \) and the longer the path is the lower the chance of false positives is. Since \( P \) is provided by the plaintiff, we have control over the source code and the compilation process to make sure that \( P \) would not contain a large number of dummy instructions that can increase the chance of false positives by generating significant noise in the plaintiff paths. Therefore, we define the path containment score of VDG(\( P \)) and VDG(\( S \)) as follows,

\[
\text{PCS}(\text{VDG}(P), \text{VDG}(S)) = \sum_{i=1}^{n} \omega_i \cdot \text{PCS}_{\text{path}}(p_i, \text{VDG}(S)),
\]

where \( \mathcal{P} \) is the set of paths selected from VDG(\( P \)), \( |\mathcal{P}| \) is the number of paths in \( \mathcal{P} \), \( p_i \) is the \( i \)-th path of \( \mathcal{P} \), and \( |p_i| \) is the length of path \( p_i \). \( \omega_i \), the weight of \( i \)-th path is defined as

\[
\omega_i = \frac{|p_i|}{\sum_{k=1}^{n} |p_k|}
\]

4.4 Implementation

We implemented our Value Sequence Extractor mainly inside the decoder module of QEMU 0.9.1.

QEMU improves execution speed mainly through two unique features named block translation and translation block cache. Instead of interpreting each instruction, QEMU translates each basic block on-the-fly into a block of instrumented machine code. By directly running translated machine code, QEMU gains speed over pure interpretation methods. Moreover, a cache containing once translated code blocks prevents QEMU from re-translating the same code many times. For a
rapid prototyping, current implementation of Value Sequence Extractor sacrifices above benefits.

VaPD is by nature an off-line plagiarism detection tool, so in real world VaPD will be given plenty of time to do its job. Nevertheless, it is still worthwhile to measure the processing time of VaPD. In our experiments, VaPD typically took several seconds to fifteen minutes to analyze one program. Figure 4.6 shows average performance of VaPD’s Value Sequence Extractor in a 2.0 GHz Intel quad-core Linux machine through the test programs used in this study. Our measurement indicates that a QEMU without block translation and translation block cache (QEMU/single_step) is nearly 56 times slower than original QEMU. This is mainly because we disable block translation and translation block cache, which can be enabled after we embed taint updating code in each translated code block. Also, frequent disk access to log extensive runtime information (VaPD/no_taint) introduces additional speed reduction of 7.5 times in the performance of QEMU/single_step. This is solvable by removing extra information from dump files and storing dump files in the main memory. Performance overhead of our taint analyzer (VaPD/taint) is shown to be insignificant.

4.5 Experiment

During our evaluation of the prototype, we answer three questions. First, how resilient is VaPD to obfuscation techniques? Second, how likely will it make a false accusation? Finally, how credible is VaPD when tested with very similar programs independently implemented to meet the same specification? We thoroughly test obfuscation resiliency of VaPD using the obfuscators implemented in SandMark
Figure 4.7. Similarity scores (y-axis) of original JLex to obfuscated ones and other programs (x-axis)

Zelix Pty Ltd.’s KlassMaster [40], and Semantic Designs Inc.’s Thicket C obfuscator [75]. SandMark and KlassMaster are Java bytecode obfuscators: The latest SandMark includes 15 application obfuscations, 7 class obfuscations, and 17 method obfuscations; Zelix Pty Ltd. claims KlassMaster is a heavy duty obfuscator implementing name obfuscation, comprehensive flow obfuscation techniques, and string encryption. The Thicket C obfuscator is a C source code rewriting tool based on abstract syntax tree. It performs several obfuscation techniques including identifier scrambling, format scrambling, replacing/simplifying statements, loop rewriting, and rewriting if-then-else conditionals [78]. Because VaPD analyzes x86 machine code, we convert Java byte code (used in SandMark and KlassMaster experiments) to x86 executable using GCJ 4.1.2, the GNU ahead-of-time Compiler for Java. As a front-end of GCC, GCJ benefits from GCC’s optimization features. We also examine VaPD’s credibility by deliberately using programs that are similar to but disparate from each other. Experiments are performed on a Linux machine equipped with an Intel Quad-Core 2.00 GHz CPU and 4GB RAM.

4.5.1 Case Study I: Obfuscation Tools

We evaluated resiliency of VaPD against advanced obfuscation techniques of SandMark and KlassMaster. Since SandMark and KlassMaster are Java bytecode obfuscators, we selected JLex [79], a lexical analyzer generator written in Java, as
the subject of our tests. In this case study, we set up two cases of experiments: a single-obfuscation experiment, where only one obfuscation technique is applied at a time, and a multiple-obfuscation experiment, where multiple obfuscators are applied to one program at once. As a dynamic analysis based solution, VaPD may not reliably identify (non-)plagiarism based on a single high similarity score. Hence, in this experiment, we used 20 different inputs and compute the average similarity scores.

4.5.1.1 Impact of Single Obfuscation

In single-obfuscation experiments, original JLex is compared to obfuscated versions of itself. Also, we compare JLex to 11 additional programs (bzip2, cksum, gzip, md5sum, zip, and openssl computing MD2, MD4, MD5, RMD160, SHA1, and SHA) totally different from JLex while processing the same input. The result is shown in Figure 4.7, where the $x$-axis shows suspect program names (JLex’s obfuscated versions and other programs), and the $y$-axis shows the similarity scores.

We observed that in all cases of comparing original JLex to its obfuscated versions (totally 680 comparisons, given by 34 obfuscators and 20 inputs), the similarity scores mark 1.0. In contrast, the similarity scores between JLex and 11 other programs mark very low scores with average of 0.07. Only one case mark 0.19, which is still very low considering that the similarity score for a real plagiarism is 1.0.

Therefore, the results shown in Figure 4.7 provide us with clear answers to the questions we raised earlier: Regardless of obfuscation techniques, VaPD computed noticeably high similarity scores between the original and obfuscated programs, and discernibly lower similarity scores between different programs. In all cases, VaPD can identify the identical programs with no false accusation with an appropriate threshold (say 0.90).
Table 4.4. Names of obfuscation techniques applied to JLex to generate multiply obfuscated versions

<table>
<thead>
<tr>
<th>Control obfuscation</th>
<th>Data obfuscation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transparent Branch Insertion</td>
<td>Array Folder</td>
</tr>
<tr>
<td>Simple Opaque Predicates</td>
<td>Integer Array Splitter</td>
</tr>
<tr>
<td>Inliner</td>
<td>Promote Primitive Registers</td>
</tr>
<tr>
<td>Opaque Predicates</td>
<td>Variable Reassigner</td>
</tr>
<tr>
<td>Dynamic Inliner</td>
<td>Duplicate Registers</td>
</tr>
<tr>
<td>Interleave Methods</td>
<td>Boolean Splitter</td>
</tr>
<tr>
<td>Method Merger</td>
<td>Merge Local Integers</td>
</tr>
<tr>
<td>Reorder Instructions</td>
<td></td>
</tr>
</tbody>
</table>

4.5.1.2 Impact of Multiple Obfuscation

We also notice that a plagiarist may attempt to hide plagiarism by heavily transforming a plagiarized program through a series of obfuscators. Therefore, evaluating resiliency of VaPD against multiple obfuscation techniques applied to single program is necessary. Although it is theoretically possible for a series of multiple obfuscators to transform a program, applying many obfuscators to a single program could raise practical issues of correctness of the target program and efficiency. For example, we attempted to apply all the 39 obfuscation techniques of SandMark to JLex, but after trying several obfuscation orders, only some of them could be successfully applied. To address this problem, we selected two groups of obfuscation techniques, following the classification of Collberg et al. [36]: data obfuscation and control obfuscation. By transforming JLex through each group of obfuscators, we created two multiply obfuscated programs $\text{JLex}_{\text{control}}$ and $\text{JLex}_{\text{data}}$. In summary, we could apply 8 control obfuscators and 7 data obfuscators to JLex as shown in Table 4.4. We also generated $\text{JLex}_{\text{zkm}}$ by transforming JLex through KlassMaster with the most aggressive configuration options enabled.

We compared each of $\text{JLex}_{\text{control}}$, $\text{JLex}_{\text{data}}$, and $\text{JLex}_{\text{zkm}}$ to original JLex. In all three groups of comparisons between heavily obfuscated JLex and original JLex, we observe similarity score of 1.00. This shows that VaPD is effective in detecting plagiarisms obfuscated heavily.

---

3We could not test all 39 obfuscators because some of them failed in transforming JLex.
Table 4.5. Similarity scores of four XML parsers cross compared. (P=Plaintiff, S=Suspect)

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expat</td>
<td>0.12</td>
<td>0.17</td>
</tr>
<tr>
<td>LibXML2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RXP</td>
<td>0.08</td>
<td>0.09</td>
</tr>
<tr>
<td>Xerces-C</td>
<td>0</td>
<td>0.02</td>
</tr>
</tbody>
</table>

4.5.2 Case Study II: Similar Programs

To investigate the credibility of VaPD on analyzing highly similar but disparate programs, we cross analyze four individual XML parsers: RXP, used by the LT XML toolkit and the Festival speech synthesis system; Expat XML parser, the underlying XML parser for the open source Mozilla project and Perl’s XML::Parser; Libxml2, the XML C parser and toolkit of Gnome; and Xerces-C++ supported by Apache XML project. For each of the four XML parsers, we wrote a simple test program that parses test input and prints the parser’s internal information to the terminal. We cross-compared the refined value sequences of plaintiff programs to the value sequences of suspect programs through 240 distinct comparison cases given by four plaintiff programs, four suspect programs, five different optimization switches (O0-3 and Os) for each of the programs, and three test inputs.

To our best knowledge, these four XML parsers do not share code. Since they are all individually developed projects, it would be a false accusation if VaPD computes a higher similarity score (say, greater than 0.9) for any of them. Average and standard deviation of similarity scores of 60 cases comparing same programs are 1.0 and 0 respectively. Average and standard deviation of 180 cases comparing different programs are both 0.06. Table 4.5 summarizes the results (We show only average of similarity scores per each program pair for brevity). In all cases comparing different programs, except one case, we observe similarity scores lower than 0.17. Only one comparison case shows a similarity score of 0.23, which is still very low. Therefore, it is safe to say VaPD claims no false accusation in this case study.
4.5.3 Case Study III: Different Programs

Previously, at the end of Section 4.2, we presented preliminary results on the likelihood of VaPD raising false accusations by cross-comparing bzip2, gzip, and oggenc. In this section, we investigate even further by comparing each of bzip2, gzip, and oggenc against 9 of 11 programs used in Section 4.5.1.1—two are excluded because they overlap bzip2 and gzip. Bzip2, gzip, and oggenc used in this experiment are compiled from self-contained, single compilation-unit C programs [80], therefore they need no external libraries other than the standard C library.

From 270 distinct comparisons given by three plaintiff programs (bzip2, gzip, and oggenc), 9 suspect programs (cksum; md5sum; openssl computing MD2, MD4, MD5, RMD160, SHA1, and SHA; and zip), and 10 input files, we observe similarity scores between 0 and 0.27 except the cases of zip and gzip pairs in which all the similarity scores are 1.0. According to the documentations of zip and gzip projects, we found that zip and gzip are based on the same compression algorithm deflate which is also implemented in the zLib library. Our source code analysis confirms that the gzip used in this experiment contains code from zLib 1.1.4 in itself, and the zip is dynamically linked to the system-wide zLib 1.2.3. Therefore, high similarity scores of zip and gzip pairs are not false positives. Rather, it gives more credential to VaPD’s detection. In addition, zip scored very low similarity scores (0.01 to 0.03) against bzip2 that is also a compression utility. This result is also correct because bzip2 uses a different compression algorithm called block sorting.

4.5.4 Evaluation of the VDG Based Method

As we discussed in Section 4.3.5, VaPD can be extended to use the VDG based plagiarism detection. First, we performed the same experiments shown in Section 4.5.1 and 4.5.2 to compare the results of the VDG based method with the results of the LCS based method. As shown in Figure 4.8, with an error of less than 0.6, VDG based method computes similarity scores very close to the results of the LCS based method presented in Figure 4.7. Table 4.6 shows the results of the VDG based method applied to the same experiment shown in Section 4.5.2. The VDG-based method computed similarity scores lower than 0.27 for the cases comparing different programs, which leads to the same conclusion as the one presented
Figure 4.8. VDG-based similarity scores (y-axis) of original JLex compared to obfuscated ones and other programs (x-axis). Top 10% longest paths are compared.

Table 4.6. VDG-based similarity scores of four XML parsers cross compared. Top 10% longest paths are compared. (P=Plaintiff, S=Suspect)

<table>
<thead>
<tr>
<th></th>
<th>Expat</th>
<th>LibXML2</th>
<th>RXP</th>
<th>Xerces-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expat</td>
<td>0.04</td>
<td>0.22</td>
<td>0.27</td>
<td>0.22</td>
</tr>
<tr>
<td>LibXML2</td>
<td>0.11</td>
<td>0.26</td>
<td>0.18</td>
<td>0.14</td>
</tr>
<tr>
<td>RXP</td>
<td>0.04</td>
<td>0.13</td>
<td>0.18</td>
<td>0.27</td>
</tr>
</tbody>
</table>

in Section 4.5.2.

Second, to see the impact of a large VDG, we evaluate the performance through running a test case where each VDG contains more than a million edges. We perform path containment test on VDGs constructed from three versions of GCC (3.4.6, 4.1.2, and 4.3.2) compiling a long C source file (xlfun2.c) from XLisp source tree. The results are shown in Table 4.7, where $P$ is plaintiff program, $S$ is suspect program, $E_P$ and $E_S$ are the number of edges in VDG($P$) and VDG($S$) respectively, $E'_S$ is the number of edges in VDG($S$) after VDG reduction, $|P|$ is the number of paths in VDG($P$), $t$ is the elapsed time when testing entire paths, and $t_{10}$ is the elapsed time when testing the top 10 percent longest paths. Note that the only reason why we tested all the paths of VDG($P$) generated from a long input is to see the impact on the performance. In this experiment, we could reach the same conclusion by testing only the top 10% longest paths obtained from a smaller input file.
Table 4.7. Number of edges and elapsed time of the VDG based method comparing GCC versions.

| P   | S   | $E_P$ | $E_S$ | $E_S'$ | |P| | $t$(min) | $t_{10}$(min) |
|-----|-----|-------|-------|--------|----|--------|--------|--------------|
| 3.4.6 | 4.1.2 | 1.7M  | 2.0M  | 0.3M   | 215K | 276    | 90     |
| 3.4.6 | 4.3.2 | 1.7M  | 1.5M  | 0.3M   | 215K | 273    | 130    |
| 4.1.2 | 4.3.2 | 1.9M  | 1.5M  | 0.3M   | 90K  | 92     | 55     |

In addition, GCC 3.4.6 uses a Bison generated parser, and it was completely rewritten into a hand-written recursive descent parser since GCC 4.1.0 was released. Our path-containment scores confirm this fact: The containment scores of GCC 3.4.6 against GCC 4.1.2 and 4.3.2 are 0.68 and 0.72 respectively; and the containment score of GCC 4.1.2 against 4.3.2 is 0.92.

4.6 Discussion

4.6.1 Obfuscation Transformations and Attacks

As we discussed in Section 4.3.5, VaPD is robust against instruction/block reordering and multi-threading. In addition, we discuss about the impact of data obfuscation and potential attacks to VaPD in this section.

4.6.1.1 Data Transformation

Simple data transformations expose the core-values of the original program. Figure 4.9(a) is an example where the original values of $x$ are transformed by adding a constant. Assuming that $x$ is tainted and is 10 at the beginning, the value sequence of the transformed code is \{11, ..., 10, ..., $y$, 10\}. In this sequence, \{..., 10, ..., $y$\} are the values captured from intermediate data for computing $y$, and this must appear in the value sequence of the original code as well. Let us look at a more complex example, variable encoding transformation. In general, variable encoding transforms variable $v$ to $\alpha v + \beta$. In Figure 4.9(b), variable $y$ at line 7 of the original code is transformed to be $y + x$. For this transformation, we apply the same procedure as Drape et al. used [81]. Assuming that $y$ is tainted at line 2, the refined value sequence that VaPD extracts from the original code is \{1, 2, 6, 24\}, and the value sequence extracted from the transformed code is \{2, 1, 1,
001: x = 10;
002: x = x + 1;
003: y = ... (x - 1) ... ;
004: x = x - 1;
005: out(x);
006: out(y);

(a) Simple Data Transformation

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Transformed Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>001: x = 0;</td>
<td>001: x = 0;</td>
</tr>
<tr>
<td>002: y = 1;</td>
<td>002: y = 1;</td>
</tr>
<tr>
<td></td>
<td>// y is tainted</td>
</tr>
<tr>
<td>003: i = 0;</td>
<td>003: i = 0;</td>
</tr>
<tr>
<td>004: while (i &lt; 5) {</td>
<td>004: while (i &lt; 5) {</td>
</tr>
<tr>
<td>005: i ++ ;</td>
<td>005: i ++ ;</td>
</tr>
<tr>
<td>006: x = x + i ;</td>
<td>006: x = x + i ;</td>
</tr>
<tr>
<td>007: y = i * i ;</td>
<td>007: y = (y + i - x) * i + x ;</td>
</tr>
<tr>
<td>008: }</td>
<td>008: }</td>
</tr>
<tr>
<td>009: out(y) ;</td>
<td>009: out(y - x) ;</td>
</tr>
</tbody>
</table>

(b) More Complex Data Transformation

Figure 4.9. Data Transformation Examples. Underlined codes are added by the transformation.

2, 4, 1, 2, 5, 8, 2, 6, 12, 16, 6, 24, 34, 39, 24, 120, 135, 120, 24}. Again we see some of the encoded values are restored to original data at some points during the execution. Splitting a variable and merging two variables into one also have similar characteristics. Those invariant values are very close to the core-values of the original program, and will be included in the values extracted by VaPD.

4.6.1.2 Noise Injection (Inserting Arbitrary Instructions)

In VDG based detection, VaPD will not be affected unless a large amount of edges are removed from the VDG of a program. Inserting bogus instructions will only add nodes and edges, therefore it will not change VaPD’s detection results. Under either of the LCS and VDG metrics, injection of a huge amount of noise might increase the similarity score. If a naive program happens to generate many noisy values, this will raise the chance of false accusation. However, for malicious programs that try to hide their plagiarisms, intentionally injected noise values will result in a higher chance of being accused. Therefore, if a plagiarist comes
to know the mechanism of VaPD, he will never try to evade VaPD by injecting random noise. Moreover, automated noise injection is difficult because if the noise is not tainted, it will be filtered due to our dynamic taint analysis. However, if injected successfully, noise could dramatically increase the size of an extracted value sequence, thus slowing down the similarity score computation, consuming more memory space. We will consider sliding over a stream of values so that we may keep only a small portion of a value sequence in memory during runtime.

4.6.1.3 Loop Rewriting

Another possible counterattack is rewriting a loop in a reverse order. However, automatic loop reversing is very difficult because they could result in semantically different programs. So far, we are not aware of such tools to our best knowledge. Although some specific types of loops that are not tightly bound with the loop counters could theoretically be reversed, reversing the loop counter variable only will not affect the whole value sequence because we can eliminate values produced by loop counters (by dynamic taint analysis). One might manually reverse a loop, if at all possible, but its impact could be very limited in a large program.

4.6.2 Core-Part Plagiarism

Core-part plagiarism is a harder problem. In such case, only some part of a program or software product is plagiarized. For example, a less ethical developer may steal code from some open source projects and fit the essential module into his project with obfuscation. Let \( I_{PM} \) and \( I_{SM} \) be the input to the plagiarized module and suspect module respectively, and \( \mathcal{V}(x) \) be a value based characteristic such as a value sequence or a value dependence graph extracted from \( x \), a program or a module. Then both the value sequence based method and the VDG based method can be applied to a subproblem of the core-part plagiarism detection where \( I_{PM} = I_{SM} \). In this case, we can directly search in \( \mathcal{V}(S) \) of the suspect program, for \( \mathcal{V}(PM) \) of the plaintiff module to check whether the module has been plagiarized. For example, in the case of web browser layout engine plagiarism, given an input URL \( I \), we can first obtain \( \mathcal{V}(PM) \) from the plaintiff layout engine module; then, using the same input \( I \) we can obtain \( \mathcal{V}(S) \) from the suspect program. If the
plaintiff program and the suspect program use the same layout engine, then \( V(\text{PM}) \) and part of \( V(\text{S}) \) \( (i.e., V(\text{SM})) \) to bear significantly similar patterns. Therefore, we can search for \( V(\text{PM}) \) in \( V(\text{S}) \). Since our path-containment metric provides a better scalability than our LCS based metric, the VDG based method is more suitable for this scenario.

### 4.6.3 Limitations

Our technique bears the following limitations. First, the discussion on the core-values is largely in the abstract since this concept is difficult to be formalized or proven. A more comprehensive study of core-values is on the list of future work, however, it is not necessary for the scope of this thesis.

Second, besides the ability of extracting value sequences from the entire scope of the plaintiff program, VaPD provides the partial extraction mode in which it can extract value sequences from only a small part of the program. Based on this, we discuss about the feasibility of applying VaPD to the partial plagiarism detection problems in Section 4.6.2. However, we have not yet comprehensively evaluated this issue with real world test subjects.

Third, VaPD will not apply if the program implements a very simple algorithm. In such cases, the value sequences can be too short, which increases sensitivity to noises. Our metric is likely to cause false positives when a very short value sequence is compared to a much longer one.

Fourth, as a detection system, there exists a trade-off between false positives and false negatives. The detection result of our tool depends on the similarity score threshold. Unfortunately, without many real-world plagiarism samples, we are unable to show concrete results on such false rates. As such, rather than applying our tool to “prove” software plagiarisms, in practice one may use it to collect initial evidences before taking further investigations, which often involve nontechnical actions.

Fifth, built upon a dynamic taint analyzer, VaPD may generate much shorter value sequences if tainted data is used as an index into a translation table, or a plagiarist attempts anti-taint-analysis techniques [17]. Anti-taint attacks and some countermeasures \( (e.g., \) tainting the program counter when a tainted conditional is
tested [82]) are well summarized by Cavallaro et al. [83].

4.7 Conclusion

Obfuscation resilient code characterization is important for many code analysis applications, including software plagiarism detection. Prior techniques have limited applicability in software plagiarism detection because they either require source code analysis, cannot handle automated obfuscation tools, or rely on features of specific languages or operating systems. Motivated by an observation that some values computed by machine instructions survive various semantics-preserving code transformations, we have proposed techniques to extract and refine runtime values from the outcome of machine instructions. Leveraging such invariant values, our technique is resilient to various control and data obfuscation techniques. To our best knowledge, our work is the first one to explore the existence of core-values and use them to characterize code fragments. We have evaluated the proposed techniques in the context of software plagiarism detection. The proposed approach directly examines executable files and does not need to access the source code of suspicious programs. We analyzed some real world programs including four XML parsers, bzip2, gzip, bzip2, oggenc, openssl, zip, and JLex along with a comprehensive set of obfuscation techniques in SandMark, KlassMaster, Thicket, and Loco/Diablo. We also provide a preliminary experiment results on detecting plagiarisms heavily transformed through multiple obfuscation techniques. Our results show that the value-based method is effective in identifying software plagiarism.
Chapter 5

Conclusion

Non-legitimate use of code is a serious threat to the computer and information security either as itself or as a cause of various detrimental activities. In this study, two critical security measures against non-legitimate use of code are investigated. First, we propose PWC, the Proactive Worm Containment technique to defend against self-propagating malicious code. PWC is, to the best knowledge of the authors, the first worm containment system that effectively uses proactive contain-and-relax framework. In contrast to the worm alerts of conventional worm containment systems that have extremely low false positive rates, PWC uses a very early but immature (or inaccurate) worm alert called infection indicator. Because the indicator is available much earlier than conventional worm alerts, PWC can reduce the number of worm scans that escape the containment system during the detection delay period. The impact of false positives in the indicators is well mitigated through the white-host detection and the relaxation analysis. White-host detection prevents false indicators from initiating containment at uninfected hosts. Containment nevertheless initiated at an uninfected host is quickly deactivated through the relaxation analysis. We evaluated PWC through extensive simulations using several days of real world traffic collected at the border of University of Auckland. PWC significantly outperforms Virus Throttle and Hamsa in terms of the number of released worm scans and the number of hosts infected by local scans. The collateral denial-of-service effect of PWC is negligible compared to that of Virus Throttle.

Second, we propose a Value-Based Approach to Software Plagiarism Detection
(VaPD) to defend against code theft, that is, an illegal use of program code developed by others. VaPD detects software plagiarism without requiring the source code. Through dynamic analysis implemented in a virtual machine, VaPD extracts the value sequence, a novel dynamic and invariant behavioral semantic indicator, directly from executable files. After refining the value sequences, VaPD computes LCS-based similarity scores of two value sequences to detect plagiarism. VaPD also can organize the values in a refined value sequence into a value dependence graph (VDG) showing the data dependence between the values to handle reordering attacks and the non-determinism caused by multi-threaded programs. With VDGs from the plaintiff and the suspicious program, VaPD applies the path-containment test to determine the similarity between the programs represented by the VDGs. Experimental results show that VaPD is resilient to more than 38 code obfuscation techniques implemented in SandMark, commercial obfuscators such as Thicket and Zelix KlassMaster, Control Flow Flattening, different code optimization techniques implemented in GCC and Watcom C Compiler, and the code generation of different compilers. The effectiveness of VaPD is not limited to the whole-program plagiarism. It can also be used to detect certain cases of partial plagiarism. As a generic semantic indicator, the value sequence has a broad applicability in software analysis. For example, VaPD is also applicable for identifying and classifying malcode variants that are obfuscated through metamorphism and/or polymorphism.

By providing measures to suppress epidemics of computer worms and discourage detrimental activities related to code plagiarism, these two techniques contribute to maintenance of uninterrupted and reliable communications for the Internet and promote a more healthy and trustworthy environment for the software industry. The technique proposed in Chapter 3 has been published in the Security and Communication Networks journal [84], and the preliminary result is published in the 3rd International Conference on Security and Privacy in Communication Networks (SecureComm) 2007 [85]. PWC is also evaluated on DETER test bed, and the result is published in the DETER Community Workshop [86]. The study discussed in Chapter 4 has been submitted to 18th Annual Network & Distributed System Security Symposium for review.
5.1 Future Work

5.1.1 Proactive Worm Containment

PWC is a proactive and collaborative worm containment system, which is based on propagation of the infection indicator. The infection indicator is an immature alert that is available much earlier than a relatively trustworthy worm alert in traditional worm containment systems, which makes PWC react much earlier than conventional worm containment systems do. The availability loss caused by initial containment is mitigated to an acceptable level (i.e., lower than that of the Virus Throttle) through the white-host detection and the relaxation analysis. A reason of the availability loss is the false infection indicators. A false indicator can lead to availability loss at uninfected hosts by initiating active containment at the host where infection was detected, and by starting passive containment at the hosts to which the indicator is propagated. Therefore, we believe improving detection and propagation of infection indicator can help PWC provide a better availability. We approach to this issue in the following two ways.

First, we can increase accuracy of the indicator detection by automatically adjusting parameters at a PWC agent. Note that Figure 3.6 shows that the number of containments initiated at an uninfected host \( (rAC \text{ and } rPC) \) decreases as the indicator detector monitors more scan suspects \( (N) \) to compute the outbound connection rate. We can develop a way for a PWC agent to measure *the accuracy factor* of its indicator detection, and based on the accuracy factor, the PWC agent can automatically adjust its parameters to achieve a better performance. For example, if a PWC agent evaluates its indicator detection to be accurate enough, the PWC agent may decrease \( N \) to detect possible worm activities quicker. If a PWC agent recognizes that the accuracy of its indicator detection is below a certain threshold, it can increase \( N \) to get a better accuracy at a cost of a longer detection delay.

Second, we can reduce the number of passive containments performed at an uninfected host by filtering out propagated indicators that are likely to be false positives. Suppose that a host protected by a PWC agent runs a legitimate program showing worm-like behavior (i.e., P2P file sharing programs). The host’s PWC agent may detect and propagate false infection indicators more frequently
than a normal host would. As a result, the number of passive containments performed at uninfected hosts can be greatly increased. We believe that improving propagated indicators to carry the accuracy factors of their initiators will enable the PWC agent to determine trustworthiness of received indicators. When the accuracy factor of a received indicator is lower than that of local PWC agent, the PWC agent can give priority to its local detector and ignore the propagated indicator.

5.1.2 Value-Based Software Plagiarism Detection

In this section, we summarize current issues about VaPD, and propose the solutions as future work. First, we will examine the relationship between values. In this study, we exploited data dependence between values to create value dependence graphs to enhance robustness against the reordering attacks. A better understanding of the logical connection among the values will also enable us to further remove system noise or less significant values.

Second, the problems in handling partial plagiarism is still an open problem. In this study, we considered only a particular case of the partial plagiarism detection in Section 4.6.2, where the input to the original module is the same as the input to the counterpart in the suspicious program. However, in some practical cases, it can be a problem to manage the inputs to the plaintiff module and to the suspicious module to be identical. For example, a software plagiarist may illegally use a real time computer vision library as a part of their motion recognition software, whereas the original program uses the library for different purposes, say face recognition. Related with this, I am particularly interested in the impact of inputs on value sequences and VDGs.

Third, different metrics to assess similarity of value sequences will be investigated. In this study, we exploited two metrics, the length of the LCS of two value sequences and the path containment test between two value dependence graphs. However, these two methods have their own weaknesses while having strengths over other methods. For example, our metric based on the longest common subsequence algorithm might be too sensitive to obfuscation techniques transforming order of instructions. Similarly, our path containment test shows reasonable per-
formance with one to two million edges, however it might be too expensive for comparing huge graphs containing tens of millions of edges. String similarity metrics are well studied in the field of information retrieval. I believe VaPD can be improved through a study on diverse string similarity metrics including N-Gram, Overlap Coefficient, Euclidean Distance, etc.

In addition, I am interested in studying the impact of emulation-based obfuscators such as Themida and Code Virtualizer [87] on VaPD’s performance. Such obfuscators encode original program into a specially designed bytecode instruction set and run the bytecode program in an emulator. I believe the value based detection method can handle such obfuscators.
Bibliography


91


URL http://pma.nlanr.net/Traces/long/auck4.html


URL http://www.sans.org/resources/malwarefaq/w32\_blasterworm.php


# Vita

**Yoon-Chan Jhi**

<table>
<thead>
<tr>
<th>Date</th>
<th>Degree</th>
<th>Institution</th>
<th>Thesis/Project</th>
<th>Advisor</th>
<th>Area of Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/2010</td>
<td>Ph.D. in Computer Science and Engineering</td>
<td>The Pennsylvania State University, University Park</td>
<td>A Study of Selected Security Measures Against Non-Legitimate Use of Code</td>
<td>Peng Liu, Professor of Information Sciences and Technology</td>
<td>Worm Containment, Code Characterization and Identification</td>
</tr>
<tr>
<td>08/1998</td>
<td>M.S. in Computer Science and Engineering</td>
<td>Inha University, South Korea</td>
<td>A Prolog Tailoring Technique on an Epilog Tailored Procedures</td>
<td>Ki-Chang Kim, Assistant Professor of Computer Science and Engineering</td>
<td>Code Optimization</td>
</tr>
<tr>
<td>02/1996</td>
<td>B.E. in Computer Science and Engineering</td>
<td>Inha University, South Korea</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>