AN INVESTIGATION OF MECHANISMS TO MITIGATE ZERO-DAY COMPUTER WORMS WITHIN COMPUTER NETWORKS

KHURRAM SHAHZAD

A thesis submitted in partial fulfilment of the requirements of the University of Greenwich for the Degree of Doctor of Philosophy

June 2015

Declaration

I certify that this work has not been accepted in substance for any degree, and is no concurrently being submitted for any degree other than that of Doctor of Philosophy being studied at the University of Greenwich. I also declare that this work is the result of my own investigations except where otherwise identified by references and that I have not plagiarised the work of the others.

Student: Khurram Shahzad
Signature:
Date:

Supervisor: Dr. Steve Woodhead
Signature:
Date:

Acknowledgments

This work would not have been possible without the invaluable advise, encouragement, and support that I received from various people in the past six years.

First and foremost, all credits to my adviser Dr. Steve Woodhead for his unstinting moral support, guidance and encouragement in this research and life in general. He has been my leading light and a role model to look up to. And the one who stood by me in the times of crisis. My thanks are due to my second supervisor Dr. Panos Bakalis for his help and support during my PhD research.

I would like to thank University of Greenwich for funding this research. I would also like to thank my colleagues Darren Smith, Luc J Tidy and Dr. A.A. Adekunle in the Internet Security Research Lab (ISRL) for their help and support.

Finally, I would like to deeply thank my family and friends for their unconditional love, care and motivational support; I have received during the whole course of my PhD studies.

Abstract

An Internet worm replicates itself by automatically infecting vulnerable systems and may infect hundreds of thousands of hosts across the Internet in tens of minutes. The speed of propagation of a worm is significantly higher than many other types of malware, including viruses. The potential for signification damage within a short time is therefore great. Worm detection and response systems must, therefore, act quickly to identify and counter the effects of worms. In this thesis, an investigation of mechanisms to mitigate zero-day computer worms has been carried out, while defining the key research questions to answer.

This thesis presents a novel distributed automated worm detection and containment scheme, RL+LA, developed during the course of this research, that is based on the correlation of Domain Name System (DNS) queries against the destination IP address of outgoing TCP SYN and UDP datagrams leaving the network boundary, while utilizing cooperation between different communicating scheme members using a custom protocol, which has been termed Friends. To the knowledge of author, this is the first implementation of such a scheme. A set of tools i.e. a Pseudo-Worm Daemon (PWD), which provides random scanning and hit-list worm like functionality; and a Virtualized Malware Testbed (VMT) for testing of worm experiments, were also developed in order to empirically evaluate the performance of the desired countermeasure scheme, RL+LA.

A set of empirical experiments were conducted by using Pseudo-Slammer and Pseudo-Witty worms with real world attributes of Slammer and Witty worms in order to evaluate PWD. The experimental results are broadly comparable to real worm outbreak reported data. Furthermore, these results are compared with a biological epidemiological model (SI model) in order to explore the applicability of SI model to cyber malware infections in general, as well as to assess its usefulness in characterising the virulence of cyber malware. From base comparison of Pseudo-Slammer and Pseudo-Witty worm experimental

results with reported outbreak data of Slammer and Witty worms; and SI model, it is concluded that: (a) PWD can be used as an effective tool to empirically analyze the propagation behaviour of random scanning and hit-list worms and to test potential countermeasures, (b) SI model can be effectively used in characterising the virulence of random scanning worms. Another comprehensive sets of empirical experiments were also conducted by using a Slammer-like pseudo-worm on a small scale with class C networks and on class A networks by using Pseudo-Slammer and Pseudo-Witty worms with real attributes of Slammer and Witty worms, without any countermeasures and by invoking RL and RL+LA countermeasures, in order to evaluate the performance of the proposed scheme, RL+LA. The experimental results show a significant reduction in the infection speed of the worms, when the countermeasure scheme is invoked.

Contents

1 Introduction 15
1.1 Chapter Introduction 15
1.1.1 Chapter Layout 16
1.2 Problem Statement 16
1.3 Research Aim and Objectives 17
1.3.1 Research Aim 17
1.3.2 Research Objectives 18
1.4 Outline of the Thesis 18
1.5 Chapter Summary 19
2 Literature Review 20
2.1 Chapter Introduction 20
2.1.1 Chapter Layout 20
2.2 Taxonomy of Computer Worms 20
2.2.1 Definitions 20
2.2.1.1 Virus 20
2.2.1.2 Worm 21
2.2.1.3 Zero-Day Worm 21
2.2.1.4 Trojan Horse 21
2.2.1.5 Rootkit 21
2.2.1.6 Botnet 21
2.2.2 Type of Worms 22
2.2.2.1 Based on Target Finding Schemes 22
2.2.2.2 Based on Transmission Schemes 24
2.2.2.3 Based on Payloads 24
2.2.2.4 Based on Intent of Worm Developer 25
2.2.2.5 Based on Existence "In the Wild" 26
2.2.3 Major Worm Outbreaks 27
2.2.4 Wormable Vulnerabilities 30
2.2.5 Windows XP Opportunity 32
2.3 Worm Detection and Prevention Mechanisms 33
2.3.1 Resource Limiting (RL) solutions 33

2.3.1.1 Williamson's IP Throttling 34
2.3.1.2 Chen et al. Failed Connection Based Rate Limiting (FC) .. 36
2.3.1.3 Schechter et al. Credit-based Rate Limiting (CB) 38
2.3.1.4 Wong et al. DNS-Based Rate Limiting 38
2.3.2 Automatic Signature Generation (ASG) solutions 40
2.3.2.1 Autograph: Towards Automated, Distributed Worm Signature Detection 40
2.3.2.2 Automated Worm Fingerprinting using Earlybird 43
2.3.2.3 Polymorphic Worm Detection using Structural Information of Executables 44
2.3.2.4 Anomalous Payload-based Worm Detection and Signature Generation 46
2.3.2.5 PolyS: Network-based Signature Generation for Zero-day Polymorphic Worms 46
2.3.2.6 LESG: Thwarting Zero-Day Polymorphic Worms With Network-Level Length-Based Signature Generation 47
2.3.2.7 An Automated Signature Generation Method for Zero-day Polymorphic Worms Based on Multilayer Perceptron Model 47
2.3.2.8 Automated Signature Generation for Zero-day Polymorphic Worms Using a Double-honeynet 47
2.3.2.9 Efficient Hybrid Technique for Detecting Zero-Day Polymorphic Worms 48
2.3.3 Behavior Based Signature Detection (BSD) solutions 48
2.3.3.1 Network Application Architecture (NAA) - A Behavioral Approach for Worm Detection 49
2.3.3.2 DNS-based Detection of Scanning Worms in an Enterprise Network 50
2.3.3.3 ARP-based Detection of Worms within an Enterprise Network 52
2.3.4 Leap Ahead (LA) solutions 54
2.3.4.1 Cooperative Response Strategies for Large Scale Attack Mitigation 54
2.3.4.2 COVERAGE 55

2.3.4.3 Very Fast Containment of Scanning Worms 55
2.3.4.4 Monitoring and Early Warning of Internet Worms 56
2.3.5 Predesigned-Preventative (PP) solutions 58
2.3.5.1 Epidemic Profiles and Defense of Scale-Free Networks 58
2.3.5.2 Least Effort Strategies for Cyber Security 58
2.3.5.3 A Virtual Honeypot Framework 59
2.3.5.4 Building and Deploying Billy Goat, Worm-Detection System 59
2.3.5.5 Boundary Detection and Containment of Local Worm Infections 60
2.3.5.6 Defending against Hit-list Worms using Network Address Space Randomization 62
2.3.6 Mobile Combat (MC) solutions 63
2.3.6.1 Predators: Good Mobile Code Combat against Computer Viruses 64
2.3.6.2 Models of Active Worm Defense 64
2.3.6.3 Mobile Combat /Beneficial worms "In the Wild" 64
2.3.7 Hybrid Quarantine Defense (HQD) solutions 65
2.3.7.1 A Hybrid Quarantine Defense 65
2.3.8 Defensive Worms (DW) solutions 67
2.4 Worm Testing Environments 68
2.4.1 Physical Network Testbeds 68
2.4.2 Simulation Testbeds 68
2.4.3 Emulation Testbeds 69
2.4.4 Full System Virtualization Testbeds 69
2.5 Mathematical Models for Worm Propagation 70
2.6 Research Questions 71
2.6.1 Research Question 1 71
2.6.2 Research Question 2 72
2.7 Chapter Summary 72
3 The Rate Limiting + Leap Ahead (RL+LA) Scheme 74
3.1 Chapter Introduction 74
3.1.1 Chapter Layout 74
3.2 Basic Design and Methodology 75

3.3 The RL+LA: System Design and Implementation 76
3.4 Chapter Summary 79
4 The Pseudo-Worm Daemon (PWD) 80
4.1 Chapter Introduction 80
4.1.1 Chapter Layout 80
4.2 Basic Design and Methodology of Pseudo-Worm Daemon (PWD) . 81
4.3 Pseudo-Worm Daemon System Design and Implementation 81
4.4 Characteristics of the Pseudo-Worm Daemon (PWD) 85
4.4.1 UDP based Propagation 85
4.4.2 Pseudo Random Number Scanning 85
4.4.3 Hit-List 85
4.4.4 Containment 86
4.4.5 Scanning rate 86
4.4.6 Authentication 86
4.4.7 Logging and Reporting 86
4.5 Evaluation of Pseudo-Worm Daemon (PWD) 86
4.5.1 Pseudo-Slammer Worm Experiments 87
4.5.1.1 Slammer Worm Outbreak Attributes 87
4.5.1.2 Experimental Setup 87
4.5.1.3 Experimental Methodology 89
4.5.1.4 Experimental Results 89
4.5.2 Pseudo-Witty Worm Experiments 90
4.5.2.1 Witty Worm Outbreak Attributes 90
4.5.2.2 Experimental Setup 90
4.5.2.3 Experimental Methodology 92
4.5.3 Discussion 93
4.5.3.1 Empirical Analysis of Pseudo-Slammer Worm Results 93
4.5.3.2 Empirical Analysis of Pseudo-Witty Worm Results 93
4.5.4 Epidemiological Modelling 95
4.5.4.1 Classical Simple Epidemic Model 95
4.5.4.2 Modeling Methodology and Results 96
4.6 Virtualized Malware Testbed (VMT) 97
4.6.1 Introduction 97
4.6.2 VMT Architecture Design and Implementation 98

4.6.3 Characteristics of the Virtualized Malware Testbed (VMT) ... 101
4.6.3.1 Scale ... 101
4.6.3.2 Cost ... 101
4.6.3.3 Flexible and Efficient Worm Experiment Control ... 102
4.6.3.4 Isolation ... 102
4.6.3.5 Remote Administration ... 102
4.6.3.6 Confinement ... 102
4.7 Chapter Summary ... 102
5 Experimental Results for The RL+LA Scheme on a Small Scale Network. ... 104
5.1 Introduction ... 104
5.1.1 Chapter Layout ... 104
5.2 Experimental setup ... 104
5.3 Experimental Methodology ... 105
5.4 Experimental Results ... 107
5.5 Discussion and Future Work ... 110
5.6 Chapter Summary ... 112
6 Experimental Results for The RL+LA Scheme on Class A Scale Networks with Real Worm Outbreak Attributes ... 113
6.1 Introduction ... 113
6.1.1 Chapter Layout ... 113
6.2 Pseudo-Slammer Worm Experiments ... 114
6.2.1 Slammer Worm ... 114
6.2.2 Experimental Setup ... 114
6.2.3 Experimental Methodology ... 116
6.2.4 Experimental Results ... 117
6.2.4.1 No Countermeasure ... 117
6.2.4.2 RL Countermeasure ... 117
6.2.4.3 RL+LA Countermeasure ... 118
6.3 Pseudo-Witty Worm Experiments ... 120
6.3.1 Witty Worm ... 120
6.3.2 Experimental Setup ... 120
6.3.3 Experimental Methodology ... 122
6.3.4 Experimental Results ... 122

6.3.4.1 No Countermeasure 122
6.3.4.2 RL Countermeasure 123
6.3.4.3 RL+LA Countermeasure 123
6.4 Discussion 125
6.4.1 Comparison of Pseudo-Slammer Worm Results 125
6.4.2 Comparison of Pseudo-Witty Worm Results 127
6.4.3 Alternative Network Topologies 130
6.4.4 RL+LA Countermeasure Overhead 130
6.4.5 Applicability of RL+LA Experimental Results on the Internet Scale 130
6.5 Chapter Summary 130
7 Conclusions 132
7.1 Chapter Introduction 132
7.1.1 Chapter Layout 132
7.2 Summary of Suggested Original Contributions 132
7.2.1 Research Question 1 132
7.2.2 Research Question 2 133
7.3 Recommendations for Future Work 134
7.3.1 The Rate Limiting + Leap Ahead (RL+LA) Scheme 134
7.4 List of Publications 134
7.4.1 Published Papers 134
7.5 Chapter Summary 135
8 Bibliography 136
9 Appendices 146
9.1 The RL+LA Source Code 146
9.2 Pseudo-Worm Daemon (PWD) Source Code 165

List of Figures

Figure 2-1 Williamson IP throttling implementation in Windows (Williamson, 2002) 35
Figure 2-2 Distributed Anti-Worm Architecture (Chen and Tang, 2004) 37
Figure 2-3 Cascading bucket Rate limiting scheme (Wong et al., 2005) 39
Figure 2-4 Architecture of an Autograph Monitor (Kim and Karp, 2004) 41
Figure 2-5 DNS Anomaly-based Detection Deployment (Whyte, Kranakis and Oorschot, 2005) 51
Figure 2-6 ARP-based Detection of Worms within an Enterprise Network (Whyte, van Oorschot and Kranakis, 2005) 54
Figure 2-7 Worm Monitoring System (Zou et al., 2003) 57
Figure 2-8 The "unreachable destination" Behaviour using the RBG Architecture (Zamboni, Riordan and Yates, 2007) 61
Figure 2-9 Connection Rate Limitations and Friends Overview (Porras et al., 2004) 66
Figure 3-1 The RL+LA Proposed Design Architecture 77
Figure 3-2 Flow Chart for The RL+LA Prototype Algorithm 78
Figure 4-1 Worm Infection Process (Chen and Robert, 2004) 82
Figure 4-2 Design Architecture of PWD 83
Figure 4-3 Flow Diagram of PWD Algorithm 84
Figure 4-4 Slammer Worm Experimental Test Network 88
Figure 4-5 Experimental Results of Pesudo-Slammer Worm 90
Figure 4-6 Witty Worm Experimental Test Network 91
Figure 4-7 Experimental Results of Pesudo-Witty Worm 92
Figure 4-8 Pseudo-Slammer Experiments vs. Real Slammer Outbreak 93
Figure 4-9 Pseudo-Witty Experiments vs. Reported Witty Outbreak 94
Figure 4-10 Best Fit SI Model for Pseudo-Slammer Worm Experimental Data 96
Figure 4-11 Best Fit SI Model for Pseudo-Witty Worm Experimental Data 97
Figure 4-12 VMT Physical Network Setup 100
Figure 5-1 Experimental Test Network 106

Figure 5-2 Experimental Results with 25 % of Hosts Vulnerable to Infection 107
Figure 5-3 Experimental Results with 20 % of Hosts Vulnerable to Infection 109
Figure 5-4 Experimental Results with 15 % of Hosts Vulnerable to Infection 109
Figure 5-5 Experimental Results with 10 % of Hosts Vulnerable to Infection 110
Figure 5-6 % of Susceptible Hosts Infected for Experimental Tests 1-9... 111
Figure 6-1 Slammer Worm Experimental Test Network 115
Figure 6-2 Experimental Results of Pseudo-Slammer Worm with No Countermeasures 117
Figure 6-3 Experimental Results of Pseudo-Slammer Worm with RL Countermeasure 118
Figure 6-4 Results of Pseudo-Slammer Worm with RL+LA Countermeasure Threshold I 119
Figure 6-5 Results of Pseudo-Slammer Worm with RL+LA Countermeasure Threshold II 119
Figure 6-6 Witty Worm Experimental Test Network 121
Figure 6-7 Results of Pseudo-Witty Worm 123
Figure 6-8 Results of Pseudo-Witty Worm with RL Countermeasure 124
Figure 6-9 Results of Pseudo-Witty Worm with RL+LA Countermeasure..... 124
Figure 6-10 Comparison of Pseudo-Slammer Worm Results 126
Figure 6-11 Time of Infection for Pseudo-Slammer Experimental Tests.... 126
Figure 6-12 Comparison of Pseudo-Witty Worm Results 128
Figure 6-13 Time of Infection for Pseudo-Witty Experimental Tests 129

List of Tables

Table 2-1 Types of Worms 26
Table 2-2 Major Worm Outbreaks 27
Table 4-1 VMT Hardware and Operating System Infrastructure 100
Table 5-1 Summary of Initial Results 111
Table 6-1 Results of Pseudo-Slammer Worm 125
Table 6-2 Results of Pseudo-Witty Worm 128

1 INTRODUCTION

1.1 Chapter Introduction

Computer network worms are a very serious potential threat to computer network security due to their high potential speed of propagation and their ability to self-replicate. Zero-day worms such as SQL slammer (Moore et al., 2003) and Witty (Shannon and Moore, 2004) represent a particularly challenging class of such malware that exploits a vulnerability that has not been patched at the point of an outbreak. Such network worms are hard to prevent or contain due to their high speed of propagation and variant nature. Modern hypothetical flash worms are even capable of infecting large susceptible population of hosts on the Internet in a few seconds (Staniford et al., 2004), thereby making human mediated response for worm detection and prevention completely impractical.

Various techniques for worm detection, mitigation and containment have been proposed by researchers, such as rate limiting: Williamson's IP throttling (Williamson, 2002), Wong et al. DNS based rate limiting (Wong et al., 2005), automatic signature generation: Autograph (Kim and Karp, 2004), Earlybird (Singh et al., 2004), behaviour signature detection: DNS based detection of Scanning Worms (Whyte, Kranakis and Oorschot, 2005) and ARP based detection of worms (Whyte, van Oorschot and Kranakis, 2005), but none provide an effective and an efficient method of worm containment in the case of a fast rapid zero day worm outbreak on a large network such as the Internet.

In order to defend against such zero-day worm attacks, it is desirable to understand the propagation of worms, their propagation methods, and their detection and prevention mechanisms. Hence, the research reported in this thesis focuses on the empirical analysis of zero-day worms such as SQL Slammer and Witty, and designing and testing a potential distributed automated countermeasure for zero-day worm detection and containment, capable of automatically containing and preventing worm spread without any human intervention.

1.1.1 Chapter Layout

This chapter begins by presenting the problem statement and research question present in the domain of computer network worms in section 1.2. Section 1.3 sets out the aim and key objectives of the research reported in this thesis. Section 1.4 describes the overall structure of the thesis while section 1.5 provides the concluding statement.

1.2 Problem Statement

A network worm is a program that self-replicates and self-propagates across a network, exploiting security or policy flaws in widely-used network services, without any human intervention (Weaver et al., 2003), while zero-day worms are a type of malware that exploits a vulnerability that has not been patched at the time of the worm outbreak (Li, Salour and Su, 2008). Since the spread of the Morris worm in 1988 (Chen and Robert, 2004), computer network worms have become a persistent problem to the Internet infrastructure causing billions of dollars in losses to businesses, governments, and service providers (Chakrabarti and Manimaran, 2002). Melissa, Code Red, Blaster, SQL Slammer (also called Sapphire), Conficker etc. (Weaver et al., 2003) did considerable damage to the Internet community. SQL Slammer is considered to be the fastest random scanning worm in history as its infected population doubled in size every 8.5 seconds, with 90 % of vulnerable hosts infected within 10 minutes (Moore et al., 2003). This worm achieved its full scanning rate i.e. over 55 million scans per seconds, only 3 minutes after it was released. It did not contain any malicious payload but the amount of traffic it generated, halted small parts of the Internet for several hours. Flash, metamorphic and polymorphic worms are evolving categories of network worms, and are considered a serious threat to the Internet.

In 2004, Staniford et al. (Staniford et al., 2004) hypothesized the top speed of a properly configured flash worm. Furthermore, they predicted that a UDP worm could saturate 95% of one million vulnerable hosts on the Internet in 510 milliseconds. A similar worm using a TCP based service could saturate 95% of one million vulnerable hosts in 1.3 seconds. Today, Internet bandwidth is much greater than in 2004, whilst many constituent networks employ at least a

basic rate limiting countermeasure, with others using more sophisticated methods. It is difficult, therefore, to judge whether Staniford's figures are still accurate.

Although, there has been no major random scanning worm outbreak since the Witty event of 2004, a recent study by Tidy et al. (Tidy et al., 2014) provides a list of recent wormable vulnerabilities (a vulnerability which worms may exploit in order to propagate on the Internet), as well as highlighting the number of Windows XP hosts still connected to the Internet as documented by the Shodan search engine (SHODAN - Computer Search Engine, 2009). Some details of recent wormable vulnerabilities and the Windows XP potential threat are given in section 2.2.4 and 2.2.5 of this thesis. With the advent and increase in prevalence of cyber warfare such as Stuxnet (Falliere and Murchu, 2011), worms have again become weapon of choice for attackers, due to their fast propagation and ability to cause considerable damage on the Internet. As described previously, factors such as the availability of wormable vulnerabilities with a large number of hosts susceptible to those vulnerabilities and lack of Windows XP support with a large number of existing hosts, have increased the chances of any future potential worm outbreak.

Due to the high speed and zero-day nature of many worms, traditional intrusion detection methods (i.e. generation and deployment of attack signatures) are ineffective (Moore et al., 2003). These countermeasures also lack the ability to propagate malware warnings to uninfected sites in a timely manner. Hence, in order to effect automatic detection and containment of zero-day worms, a rapid, accurate and distributed worm detection and containment method is required.

1.3 Research Aim and Objectives

1.3.1 Research Aim

The aim of this research is to develop a worm detection and prevention mechanism that will detect and mitigate the propagation of zero-day worms.

1.3.2 Research Objectives

The following are the key research objectives which were defined at the start of the research:

1) To conduct a comprehensive literature review in the field of computer worms and their countermeasures.
2) To design, implement and empirically evaluate one countermeasure mechanism for zero-day worm detection and mitigation.
3) To design and implement suitable tools such as a pseudo-worm daemon and a virtualized testbed to allow the developed countermeasure to be empirically tested and evaluated.

1.4 Outline of the Thesis

The overall structure of the thesis is as follows: Chapter 2 presents the literature review detailing an existing taxonomy of malware, worm detection and preventions mechanisms and malware testing environments, mathematical models for worm propagation and thereby, defines the limitation of the existing research work. Chapter 3 details a distributed automated worm detection and containment scheme, termed RL+ LA (Rate Limiting + Leap Ahead) that is based on the correlation of Domain Name System (DNS) queries and the destination IP address of outgoing TCP SYN and UDP datagrams leaving the network boundary; and, cooperation between different communicating scheme members using a custom protocol, which we termed Friends. Chapter 4 describes the architecture and design of a Pseudo-Worm Daemon (PWD) having random and hit-list scanning capabilities; and details the architecture and design of the malware testing environment, Virtualized Malware Testbed (VMT), based on VMware technologies, as background to chapter 4. This chapter also presents evaluation of PWD with Pseudo-Slammer and Pseudo-Witty worms empirical experiments by comparing them with real world reported data and with an epidemiological model. Chapter 5 details the design and results of a series of empirical experiments conducted by employing the RL+LA scheme on small scale network by using PWD and VMT. The analysis of the results shows that the scheme is effective on a small scale. Chapter 6 presents the design and results of a series of empirical

experiments conducted by employing the RL+LA scheme on a Class A scale network by using a PWD (with real Slammer and Witty worm attributes) and VMT. Furthermore, this chapter also presents the detailed analysis and discussion of the experimental results. Chapter 7 concludes the thesis with the list of contributions and a list of areas of possible further research work. Finally appendices present the RL+LA and the PWD source codes.

1.5 Chapter Summary

This chapter has presented the introduction of the research domain by highlighting the research problem present within the domain of computer network worms. Furthermore, it describes the aim and objectives of the research to be undertaken while finally detailing the overall structure of the thesis. The next chapter will present the literature survey of worm detection and prevention mechanisms, worm testing environments, and the mathematical model used to describe the epidemiology of computer worm and will then set out the research questions to carry out this research.

2 LITERATURE REVIEW

2.1 Chapter Introduction

A detailed literature review was undertaken as an initial part of the work reported in this thesis, which consists of different malware concepts, worm taxonomy, key worm outbreaks, worm detection and prevention mechanisms, worm testing environments and mathematical models for worm propagation. This chapter reports the outcome of the review.

2.1.1 Chapter Layout

This chapter begins by introducing basic malware concepts and differentiates between them in section 2.2. Furthermore, it categories different worms based on the taxonomy of worms, summarizes some key worm outbreaks and their characteristics, and provides details of wormable vulnerabilities and potential threats posed by Windows XP. Section 2.3 presents and classifies different worm detection and prevention mechanisms while section 2.4 explores previously presented worm testing environments by classifying them into different classes. Section 2.5 details various mathematical models for worm propagation while section 2.6 presents the research questions developed as an outcome of sections 2.3, 2.4 and 2.5. Finally section 2.7 presents the chapter summary.

2.2 Taxonomy of Computer Worms

In order to understand the taxonomy of computer worms, first different malware (short for malicious software) related terms such as virus, worm, zero-day worm, trojan horse, rootkit and botnet need to be defined. The next sub-section introduces these terms.

2.2.1 Definitions

2.2.1.1 Virus

A computer virus can be defined as a set of program instructions that attaches itself to a file, reproduces itself and spreads to other files with the aid of human

intervention (Parsons and Oja, 2010). For example, Chernobyl virus (Symantec: W95.CIH, 1998), Bomber (F-Secure: Bomber, 1992) etc.

2.2.1.2 Worm

A computer worm is a program that self-replicates and self-propagates across a network, exploiting security or policy flaws in widely-used network services, without any human intervention (Weaver et al., 2003). For example, Code Red (Zou, Gong and Towsley, 2002), Slammer (Moore et al., 2003), Witty (Shannon and Moore, 2004) etc.

2.2.1.3 Zero-Day Worm

A zero-day worm is a type of worm that exploits a zero-day vulnerability that has not been patched or widely acknowledged at the point of exploitation (Tidy, Woodhead and Wetherall, 2013), (Weaver et al., 2003). For example, Code Red (Zou, Gong and Towsley, 2002) and Slammer (Moore et al., 2003) both exploit zero-day vulnerabilities.

2.2.1.4 Trojan Horse

A trojan horse, or trojan, is considered a malicious program that is non-self-replicating, which appears to perform a desirable function but instead also includes a malicious payload, often including a backdoor allowing unauthorized access to the target computer (CERT: Trojan Horses, 1999). For example, Beast (Beast 2.07, 2004) is a windows based backdoor program which invisibly gives full control of an infected host.

2.2.1.5 Rootkit

A rootkit is a type of malware, designed to hide the existence of certain processes or programs from normal methods of detection and enable continued privileged access to a computer. The term rootkit is a concatenation of the terms “root” (UNIX root account) and “kit” (software components which implements the tool) (McAfee, 2006). For example, Extended Copy Protection (XCP) (TIME Magazine, 2002)

2.2.1.6 Botnet

A botnet is a collection of Internet-connected programs communicating with other similar programs in order to perform various malicious tasks, such as keeping control of an Internet Relay Chat (IRC) channel, sending spam emails or participating in distributed denial-of-service attack attacks (Ramneek,

2003). The word botnet stems from the two words robot and network. For example, Storm Botnet (Holz et al., 2008).

2.2.2 Type of Worms

Worms can be classified in different ways according to target discovery schemes, transmission schemes, payloads, intents of worm developer and on the basis of worm outbreaks as follows:

2.2.2.1 Based on Target Finding Schemes

Target discovery refers to mechanisms by which a worm discovers new targets to infect. Schemes can be classified into: scanning, hit-list warhol and flash.

- ***Scanning worm:*** A scanning worm employs different scanning strategies (random, sequential, permutation etc.) to spread. Scanning refers to the process of probing a set of IP addresses to identify vulnerable hosts. For example, SQL slammer (Moore et al., 2003), Nimda (CERT: Nimda Worm, 2001), Code Red (CERT: Code Red, 2001) are random scanning worms. Following are different basic forms of scanning which a worm will employ:
    - **Sequential:** Working through an address block using an ordered set of IP addresses.
    - **Random:** Generating IP addresses out of a block in a pseudo-random fashion.
    - **Permutation:** This is a type of scanning where worm instances coordinate between themselves so that each instance scans a disjoint set of the address space.

  These basic forms of scanning can be aggregated to form more complex schemes as follows:

    - **Importance scanning worm:** A worm employing this technique spreads in two phases: in the first phase, random or routing scanning is used to build an initial distribution of vulnerable hosts and then in the second phase, it uses importance sampling technique to reduce the number of scans and attacks a large number of vulnerable hosts rapidly (Chen and Ji, 2005).

- **Topological worm:** A topological scanning worm uses an internal target list which is created by finding local information on networks such as the /etc/hosts file on UNIX hosts, or local topological information by using ARP cache tables and netstat (Weaver et al., 2003).
- **BGP scanning worm:** A BGP routing worm uses BGP scanning techniques which employ BGP routing tables to narrow the scanning addresses space. This type of worm is capable of targeting particular hosts within specific geographic location such as a specific country, ISP or autonomous system and can spreads 2 to 3 times faster than traditional random scanning worms (Zou et al., 2005).
- **Search worms/ meta-server worm:** A meta-server worm uses an externally generated target list of vulnerable hosts, which is maintained by a separate server, such as a matchmaking service's meta-server e-g. Gamespy (Gaespy Archade, 1999) or web searches using Google in order to find vulnerable targets.
- **Passive worm:** Passive worm does not scan potential victims instead it waits for target machines to contact the machine where it resides. For example; Gnuman (Eset: Win32/Gnuman, 2008), CRClean (Weaver et al., 2003) etc.
- **Hit-list worm*:*** A worm that employs a pre-generated list of vulnerable IP addresses to infect can be classified as hit-list worm, such as Witty (Shannon and Moore, 2004). Witty uses multiple spreading strategies including initial hit-list, botnet and random scanning.
- **Warhol worm:** A Warhol worm (Staniford, Paxson and Weaver, 2002) is a hypothetical very fast spreading worm that uses a combination of a hit-list (which helps initial spread) and permutation scanning (which keeps its infection rate higher than random scanning).
- **Flash worm:** Staniford et al. (Staniford et al., 2004) proposed an extension of the Warhol worm which they named the Flash worm. The flash worm contains an initial global size hit-list.

They hypothesized that a UDP based flash worm could infect 95 percent of one million vulnerable hosts in 510 ms, while a TCP based flash worm could infect the same population in 1.3s.

2.2.2.2 Based on Transmission Schemes

The transmission scheme is a mechanism that a worm employs to transmit itself to target hosts. A worm can employ either transmission control protocol (TCP) or user datagram protocol (UDP) to transmit itself. TCP is a connection oriented protocol and requires a 3-way handshake before connection establishment, while UDP is a connectionless protocol.

- ***TCP based worm:*** A TCP based worm uses transmission control protocol (TCP) as its transmission mechanism such as Code Red (CERT: Code Red, 2001). A TCP based worm tends to have greater latency than a UDP based worm as it uses a 3 way handshake for connection establishment.
- ***UDP based worm:*** A UDP based worm uses user datagram protocol (UDP) as its transmission mechanism such as SQL Slammer (Moore et al., 2003). UDP worms are bandwidth limited and are generally capable of spreading faster than TCP based worms.

2.2.2.3 Based on Payloads

Payload refers to the actual code carried by the worm apart from the propagation routines. The worm payload can be used to perform different functions such as using the target host as a spam relay as in the case of the Sobig worm (CERT: W32/Sobig.F Worm, 2003), employing the target hosts as HTML proxy as in the case of Sobig (CERT: W32/Sobig.F Worm, 2003), creating a denial of service (DOS) attack against target hosts to deny legitimate services, such as the W95/firkin.wom attack against 911 servers (McAffe: W95/firkin.worm, 2000), or cyber warfare by creating physical world damage such as Stuxnet (Falliere and Murchu, 2011) that has an ultimate goal to sabotage Iranian nuclear facilities by reprogramming programmable logic controllers (PLCs) in infected SCADA systems.

Based on worm payload itself, a worm can be classified into the following three categories (Li, Salour and Su, 2008):

- ***Monomorphic worm:*** A monomorphic worm uses a monomorphic payload that does not change during worm propagation and exhibits a consistent signature (Li, Salour and Su, 2008). A monomorphic payload can easily be detected by a signature-based detection system for non-zero day worms. Some monomorphic worms use a variable size payload in different instances by padding the payload with garbage data, but the same common signature usually applies.
- ***Polymorphic worms:*** Polymorphic worms use a polymorphic payload which changes itself by scrambling the program in different worm instances, whilst functioning in the same way. A traditional signature based detection systems will not usually detect such polymorphic worms (Li, Salour and Su, 2008). However, it may be possible to define a signature based on a common part of such a worm binary or another characteristic of the payload (see section 2.3.2).
- ***Metamorphic worms:*** Metamorphic worms use a metamorphic payload which changes itself and its behavior by using encryption in different instances of the worm (Li, Salour and Su, 2008). It is even harder to detect metamorphic worms using signature-based techniques than polymorphic worms.

2.2.2.4 Based on Intent of Worm Developer

Worms can be classified based on the intentions of the worm developer as follows:

- ***Harmful worm:*** A worm can be considered harmful if the intention of the worm writer was malicious or harmful, such as disrupting network services, physical world damage, physical world DOS, economic sabotage, terrorism, or cyber warfare etc. For example, Slammer, Code Red, Witty, Stuxnet etc. are all considered harmful worms.
- ***Beneficial worm:*** A beneficial worm, defensive worm, or anti-worm can be released with the intent of patching the vulnerabilities which can be exploited by a harmful worm. However, a beneficial worm is illegal as it patches network hosts without permission of the owner. For example, Welchia worm (Symantec: W32.Welchia.Worm, 2003), CRClean (Weaver et al., 2003) etc.

2.2.2.5 Based on Existence "In the Wild"

Worms can be classified based on worm outbreaks as follows:

- ***Existing Implemented worm:*** An existing implemented worm is one that has been released "In the Wild" on the Internet. For example, SQL slammer, Witty, Code Red etc.
- ***Hypothetical worm:*** A hypothetical worm is one that is only proposed and not released. For example, the importance scanning worm, the BGP routing worm, the flash worm and the Warhol worm.

Table 2.1 summaries the different types of worms as described in section 2.2.2.

Table 2-1 Types of Worms

<table>
<tr><th colspan="3">Type of Worms</th></tr>
<tr><td rowspan="11">Based on Target Discovery</td><td rowspan="11">Scanning worms</td><td>Random</td></tr>
<tr><td>Sequential</td></tr>
<tr><td>Permutation</td></tr>
<tr><td>Importance</td></tr>
<tr><td>Topological</td></tr>
<tr><td>BGP Scanning</td></tr>
<tr><td>Search worms/ meta-server worm</td></tr>
<tr><td>Passive worms</td></tr>
<tr><td>Hit-list worms</td></tr>
<tr><td>Warhol worm</td></tr>
<tr><td>Flash Worms</td></tr>
<tr><td rowspan="2">Based on Transmission Scheme</td><td colspan="2">TCP based worms</td></tr>
<tr><td colspan="2">UDP based worms</td></tr>
<tr><td rowspan="3">Based on Payloads</td><td colspan="2">Monomorphic worms</td></tr>
<tr><td colspan="2">Polymorphic worms</td></tr>
<tr><td colspan="2">Metamorphic worms</td></tr>
<tr><td rowspan="2">Based on Intent of Worm Developer</td><td colspan="2">Harmful worms</td></tr>
<tr><td colspan="2">Beneficial Worms</td></tr>
<tr><td rowspan="2">Based on Outbreaks</td><td colspan="2">Exiting Implemented worms</td></tr>
<tr><td colspan="2">Hypothetical worms</td></tr>
</table>

2.2.3 Major Worm Outbreaks

Table 2.2 summarizes the major worm outbreaks along with their different attributes (Li, Salour and Su, 2008), (Xiang, Fan and Zhu, 2009), (Falliere and Murchu, 2011):

Table 2-2 Major Worm Outbreaks

Major Worm Outbreaks							
Worm	**Year of Release**	**Target Finding Scheme**	**Propagation Scheme**	**Payload Format**	**Platform/ Service**	**Port**	**Vulnerability**
Morris	November 1988	Random scanning	TCP	Monomorphic	DECX, Sun 3/ sendmail, finger	25,79	Buffer overflow vulnerability
Code Red I	July 2001	Random scanning	TCP	Monomorphic	Microsoft IIS web service	80	Buffer Overflow In IIS Indexing Service DLL vulnerability
Code Red II	August 2001	Local subnet scanning	TCP	Monomorphic	Microsoft IIS web service	80	Buffer Overflow In IIS Indexing Service DLL vulnerability
Nimda	September 2001	Random scanning, Network shares, Passive	TCP , UDP	Monomorphic	Windows 95, 98, Me, NT, 2000, XP, Microsoft IIS web service	80	Microsoft IIS 4.0 / 5.0 directory traversal vulnerabilities
Slammer	January 2003	Random scanning	UDP	Monomorphic	Microsoft SQL Server 2000	1434	Buffer overflow vulnerability

Witty	March 2004	Random scanning, Hit-list,Botnet	UDP	Monomorphic	Internet Security Systems ISSs	Random destination port	ISS protocol analysis module (PAM) vulnerability
Sasser	April 2004	Second channel	TCP	Monomorphic	Windows 2000/ Security Authority Subsystem Service (LSASS)	445,9996	Buffer overflow vulnerability
Conficker	November 22, 2008	Local network scanning, Nearby other infected hosts. Random scanning,	TCP	Monomorphic	Windows 2000,XP, Server 2003, Vista, Server 2008	445	Windows Server service(MS08-067)
Stuxnet	June 2010	USB,P2P RPC, Network shares, Botnet	TCP, UDP, RPC	Monomorphic	Windows, Siemens PCS 7, WinCC and STEP7 industrial software applications that run on Windows, Siemens S7 PLCs	80 to contact C&C server	MS10-046 .LNK Vulnerability, MS10-061 Print Spooler Vulnerability, MS10-073 Win32k Keyboard Layout Vulnerability, Un-patched Task Scheduler Vulnerability, MS08-067 Windows

							Server Service vulnerability, Hardcoded username and password in WinCCMSSQL database, DLL preloading attack in Step 7 Project files, Windows rootkit to hide Windows binaries

2.2.4 Wormable Vulnerabilities

A wormable vulnerability is the vulnerability which worms exploit in order to propagate (Nazario, Ptacek and Song, 2004). According to Luc et al. (Tidy et al., 2014), a wormable vulnerability can be summarized in the Boolean equation (2.1), where a wormable vulnerability, *Vw*, is determined by not requiring human interaction, *H*, is network reachable, *Nr*, provides remote code execution, *R*, and provides network access *Na* once exploited.

$$Vw = H' \bullet Nr \bullet R \bullet Na \qquad (2.1)$$

Luc et al. reports that there are a number of resources that focus on providing details for known vulnerabilities. One such source is the Common Vulnerabilities and Exposures (CVE) system (CVE - Common Vulnerabilities and Exposures, 2014), which provide details for a range of vulnerabilities. The CVE system reports the access vector, for instance if the vulnerability is network reachable or requires human interaction, and the impact if the vulnerability were to be exploited, for instance providing remote code execution or network access. These details provide information in order to assess whether a vulnerability is wormable or not.

Luc et al. (Tidy et al., 2014) reports five wormable vulnerabilities along with their CVE code (CVE - Common Vulnerabilities and Exposures, 2014), which have the potential to be used as worm exploit on a large sale on the Internet as detailed below:

- **Microsoft Remote Desktop Protocol (RDP) - 13/03/2012 - CVE-2012-0002:** The Microsoft RDP is an application for users to remotely access window based hosts in a network. This vulnerability was present in Microsoft Windows XP SP2 and SP3, Windows Server 2003 SP2, Windows Vista SP2, Windows Server 2008 SP2, R2, and R2 SP1, and Windows 7 Gold and SP1. This vulnerability allows an attacker to send a crafted packet on port 3389 to the host running vulnerable RDP implementation, and then potentially gain remote code execution, finally allowing attacker to send copies of the malicious packet across

the network. W3Counter (W3Counter, 2014) reports that these recent editions of Windows amount to approximately 3 billion Internet connected hosts in 2012. The RDP application is disabled by default and needs to be enabled manually. One estimate for the number of RDP enabled hosts is one in every 10,000 or 3000,000 hosts (KrebsonSecurity, 2012); which is exactly similar to proportion of vulnerable hosts to the Code Red worm outbreak in 2001 (Zou, Gong and Towsley, 2002). Such a large proportion of hosts could result in a virulent worm outbreak.

- **BigAnt Message Server- 09/01/2013 - CVE-2012-6275:** The BigAnt instant messaging (IM) software is an instant messaging solution targeted towards business use. The attacker can cause buffer overflow by exploiting this vulnerability and is able to send a crafted packet to execute remote code on the targeted host. As the software links with active directly, it can lead to comprise of all user accounts and thereby, having potentially wider impact than just the host running the message server. Although this vulnerability lack the install base like Microsoft RDP vulnerability CVE-2012-0002, but this is of particular note owing to its use in a corporate setting, as well as potentially allowing access to further details which can be used to comprise hosts with active directly user accounts details and thereby allowing remote code execution on hosts. This process can also leads to create an initial hit-list of comprised hosts.
- **VMWare vCenter - 25/04/2013- VMSA-2013-0006.1:** VMWare vCenter is a management platform for VMware ESXi server running virtualised hosts. VMWare vCenter is installed on Windows Server. A number of CVEs reported under the VMWare security advisory VMSA-2013-006.1 (VMware Security Advisories, 2013) which detail how an attacker may leverage Microsoft Active Directory integration in order to gain authentication on Windows-based servers running VCenter (CVE-2013-3107), and then use this authentication in order to execute remote code using another vulnerability (CVE-2013-3079). This access grants the attacker administrative privileges to the host

system, allowing the attacker to then send copies of the malicious packets to other susceptible hosts. As VMware is being the largest vendors of software, a vulnerability in VMWare management software presents a scenario where a substantial number of management hosts may be susceptible to an attack, while also providing further access to the virtualised hosts, currently running on it. Although this vulnerability has since been patched by VMware, however, it demonstrates that virtualisation can present a potential scenario for future virulent worm outbreak.

- **ASUS RT-AC66U Router - 26/07/2013- CVE-2013-4659:** The ASUS RT-AC66U router has a vulnerability in the Broadcom ACSD service that allows an attacker to send a crafted packet on port 5916 by causing a buffer overflow attack, allowing administrative access on the target host with the ability for remote code execution and sending copies of the malicious packet to other susceptible hosts. This vulnerability demonstrates that network devices such as routers, switches etc. can also leads to potential worm outbreak.
- **systemd 208 and prior - 20/09/2013- CVE-2013-4391:** systemd is a system management service, or daemon, designed specifically for Linux-based operating systems, and forms part of the Linux start-up process. CVE-2013-4391 allow an attacker to cause buffer overflow by using a crafted packet, resulting in allowing remote code execution. In addition with another vulnerability (CVE-2013-4394),an attacker can gain administrative access, therefore allowing network access to send copies of the malicious packets to other susceptible hosts. This vulnerability demonstrates that other operating system such as Linux, aside from Windows, can also be subject to a wormable vulnerability.

2.2.5 Windows XP Opportunity

It has been estimated that Windows XP still constitutes 23.87% of all operating systems installed on desktop hosts (Net Market Share: Desktop operating system market share, 2014) while a keyword "Windows XP" search on Shodan search engine shows 7952 Windows XP live hosts, running different services, are still connected to the Internet. As of the 8th April 2014

extended support for Windows XP was discontinued, thereby, disallowing free support and security patches. Only what is termed “critical patches” will be made available to paying customers. Additionally, the built-in anti-malware tools i.e. Security Essentials and the Malicious Software Removal Toolkit will no longer be supported after the 14th July 2015. Given its lack of support, if any wormable vulnerability will exist on Windows XP, it increase the likelihood of future potential worm outbreak. This presents a particular scenario, as SQL Slammer was able to cause disruption with less than 1% of the hosts at the time being susceptible to its infection vector, therefore it is reasonable that should a Windows XP vulnerability be exploited by a Slammer-like attack, it could cause significant network disruption.

2.3 Worm Detection and Prevention Mechanisms

Worm detection and prevention has emerged as an active area of research over the last few years. Researchers have proposed various techniques for worm detection, mitigation and containment. Worm detection and prevention mechanisms can be classified into the following general categories as set out by Porras et al. (Porras et al., 2004) and Ziyad AL-Salloum (Ziyad, 2011):

- Resource Limiting (RL) or Containment solutions
- Automatic Signature Generation (ASG) solutions
- Behaviour Signature Detection (BSD) solutions
- Leap Ahead (LA) solutions
- Predesigned-Preventative (PP) solutions
- Mobile Combat (MC) solutions
- Hybrid Quarantine Defense (HQD) solutions
- Defensive Worms (DW) solutions

2.3.1 Resource Limiting (RL) solutions

Resource Limiting (RL) solutions explore ways in which local hosts or gateways may delay worm propagation through limiting the availability of resources that fast spreading worms are known to consume at high rates. For example; IP throttling (Williamson, 2002), failed-connection-based scheme

(Chen and Tang, 2004), credit-based rate limiting (Schechter, Jung and Berger, 2004), DNS based rate limiting (Wong et al., 2005).

2.3.1.1 Williamson's IP Throttling

Williamson's IP throttling scheme (Williamson, 2002) is based on the observation that during scanning worm propagation, an infected host will connect to as many different hosts as possible in unit time. An uninfected host has a different behaviour: outgoing connections are made at a lower rate, and are locally correlated (repeat connections to recently accessed hosts are likely. For example, web servers, file servers). His theory is based on the principle that restricting host-level contact rates to unique IP addresses can limit rapid connections to random addresses (e.g. worm traffic).

Williamson accomplished this by keeping a working set of addresses for each host, which models the normal contact behaviour of the host. The throttling mechanism permits outgoing connections for addresses in the working set, but delays other packets by placing them in a delay queue. If the delay queue is full, further packets are simply dropped. The packets in the delay queue are dequeued and processed at a constant rate (Williamson suggests, one per second). At the same rate, the least recently used address in the working set is removed to make room for the new connection. As a result, connections to frequently contacted addresses are allowed through with a high probability while connections to random addresses (such as those initiated by scanning worms) are likely to be delayed and possibly dropped. The size of the working set and the delay queue are important considerations for this scheme. A larger working set permits a higher contact rate while the delay queue length determines how liberal (or restrictive) the scheme is. Williamson recommends a five-address working set and a delay queue length of 100 for host-based implementations.

Williamson proposed that the worm throttle could be implemented on the Windows platform, using a similar architecture to that used by "personal firewall" software as shown in the figure 2.1. The network software of a PC has a layered architecture while the filter is best implemented as an extra layer,

or shim. Hence, all the traffic from the host can be processed by the filter. The logical way to implement the delays is to delay the initial connection attempt (e.g. the first SYN packet of the connect handshake in TCP). Since no packets will leave the host while a connection is being delayed, any networking timeouts will not be a problem. If the malicious code sets its own timeout and restarts connection attempts, these will be added to the queue.

When an host is infected by a worm which is attempting to propagate rapidly, the filter can detect this very quickly by monitoring the size or rate of increase of the delay queue. A suitable response action is then to suspend the offending application and pop up a window to alert the user. A windows service is necessary for this functionality. This has two important functions: firstly the spreading of the worm is stopped (the process in suspended); and secondly the user can (hopefully) determine whether this is a real problem or an error.

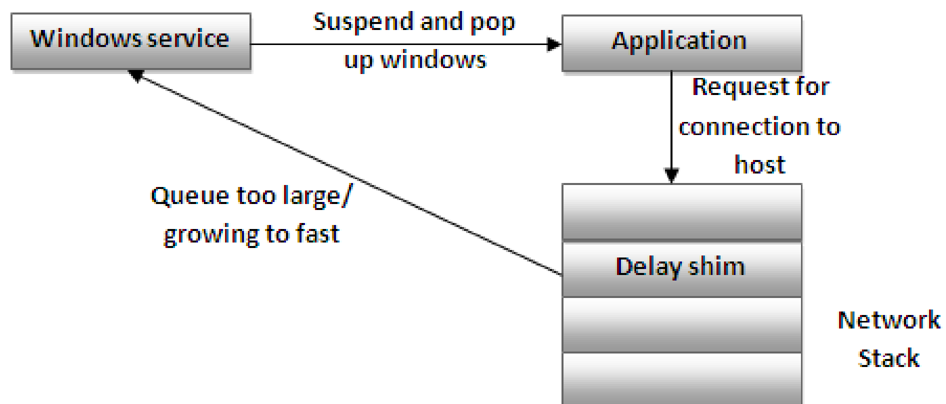

Figure 2-1 Williamson IP throttling implementation in Windows (Williamson, 2002)

Williamson's IP throttling can be deployed at an end-host as well as within an edge-based router. But, its deployment as edge-based rate limiting exhibits significantly higher false positive rates during normal operation. This is primarily due to the fact that aggregate throttling penalizes hosts with atypical traffic patterns, thereby contributing to a higher false positive rate. A possible solution is to increase the working set size at the edge to reduce the false positives, but false negatives will increase accordingly. Hence, Williamson's

throttling is best suited for end-host rate limiting where behaviour of the host is somewhat predictable.

2.3.1.2 Chen et al. Failed Connection Based Rate Limiting (FC)

Chen et al. proposed a distributed anti-worm architecture (DAW) that automatically slows down or even halts the worm propagation (Chen and Tang, 2004). Their rate limiting scheme is based on the assumption that a host infected by a scanning worm will generate a large number of failed TCP requests. When a source host makes a connection request, a SYN packet is sent to a destination address. The connection request fails if the destination host does not exist or is not listening on the port that the SYN is sent to. In the former case, an ICMP host-unreachable packet is returned to the source host; in the latter case, a TCP RESET packet is returned provided that a network firewall or router in the traffic path do not drop ICMP unreachable and TCP RESET packets. So this scheme attempts to rate limit hosts that exhibit such behaviour.

The failed connection rate limiting mechanism proposed by Chen et al. is designed to be deployed at the edge router of an ISP which consists of two software components: a DAW agent that is deployed on all edge routers of the ISP and a management station that collects data from the agents as illustrated in figure 2.2. Each agent monitors the connection-failure replies sent to the customer network that the edge router connects to. It identifies the potential offending hosts and measures their failure rates (The rate of failed connection request from a host is called the failure rate, which can be measured by monitoring the failure replies that are sent to it). If the failure rate of a host exceeds a pre-configured threshold, the agent randomly drops a minimum number of connection requests from that host in order to keep its failure rate under the threshold. Chen defined a basic rate-limit algorithm and a temporal rate-limit algorithm to constrain any worm activity to a low level over the long term, while accommodating the temporary aggressive behaviour of normal hosts.

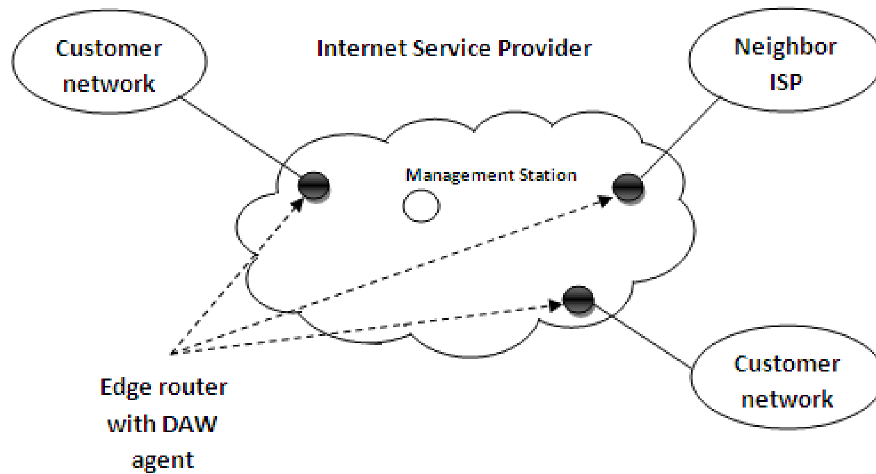

Figure 2-2 Distributed Anti-Worm Architecture (Chen and Tang, 2004)

The basic FC algorithm focuses on a short-term failure rate; λ. Chen recommends a λ value of one failure per second. Once a hash entry exceeds λ, the rate limiting engine attempts to limit the failure rate of each host in the entry to at most λ, using a leaky bucket token algorithm—a token is removed from the bucket for each failed connection and every λ seconds a new token is added to the bucket. Once the bucket for a particular host is empty, further connections from that host are dropped. Temporal FC attempts to limit both the short term failure rate λ and a longer term rate Ω. Chen suggested Ω be a daily rate and λ a per second rate. The value of Ω is intended to be significantly smaller than λ * (seconds in a day). Hosts in a hash table entry are subjected to rate limiting if the failure rate of the entry exceeds λ per second or Ω per day. The objective of temporal FC is to catch prolonged but somewhat less aggressive scanning behaviour—worms that spread under the short-term rate of λ.

Wong et al. (Wong et al., 2005) have shown from experimental data that temporal FC is more restrictive and result in higher false positives as compared to other rate limiting mechanisms (Schechter, Jung and Berger, 2004) , (Wong et al., 2005). One other limitation of Failed Connection Based Rate Limiting (FC) is that it does not address the worm activity within the local customer network. A worm-infected host is not restricted in any way from infecting other vulnerable hosts on the same customer network.

2.3.1.3 Schechter et al. Credit-based Rate Limiting (CB)

Schechter et al. (Schechter, Jung and Berger, 2004) proposed a credit-based rate limiting mechanism that is based on the observation that a worm infected host has a high rate of failed first contact connections. This technique performs rate limiting exclusively on first contact connections—outgoing connections for destination IPs that have not been visited previously while it also considers both failed and successful connection statistics. Simply described, CB allocates a certain number of connection credits per host; each failed first-contact connection depletes one credit while a successful one adds a credit. A host is only allowed to make first-contact connections if its credit balance is positive. CB maintains a Previously Contacted Host (PCH) list for each host in order to determine whether an outgoing TCP request is a first contact. Additionally, a failure credit balance is maintained for each host. Schechter suggested a 64 address PCH and a 10 credit initial balance.

Wong et al. (Wong et al., 2005) conclude that CB limits the first-contact failure rate at each host, but does not restrict the number of successful connections if the credit balance remains positive. Further, non-first-contact connections (typically legitimate traffic) are permitted through irrespective of the credit balance. Consequently, a scanning worm producing a large number of failed first contacts will quickly exhaust its credit balance and be contained. Legitimate applications typically contact previously seen addresses, and thereby are largely unaffected by the rate limiting mechanism.

2.3.1.4 Wong et al. DNS-Based Rate Limiting

Wong et al. proposed a DNS-based rate limiting mechanism (Wong et al., 2005) that is based on the rationale that worm activity shows visibly different DNS statistics from those of legitimate applications. For instance, the non-existence of DNS lookups is a tell-tale sign for scanning activity. The DNS rate limiting scheme proposed by Wong et al. states that for every outgoing TCP SYN packet, the rate limiting scheme permits it through if there exists a prior DNS translation for the destination IP address, otherwise the SYN packet is rate limited. The algorithm uses a cascading bucket scheme to contain untranslated IP datagrams. A graphical illustration of the algorithm is shown in Figure 2.3.

In this scheme, there exists a set of *n* buckets, each capable of holding *q* distinct IP addresses. The buckets are placed contiguously along the time axis and each spans a time interval *t*. The algorithm works as follows: When a TCP SYN packet is sent to an address that does not have a prior DNS translation, the destination IP address is added into the bucket for the current time interval and the packet is delayed.

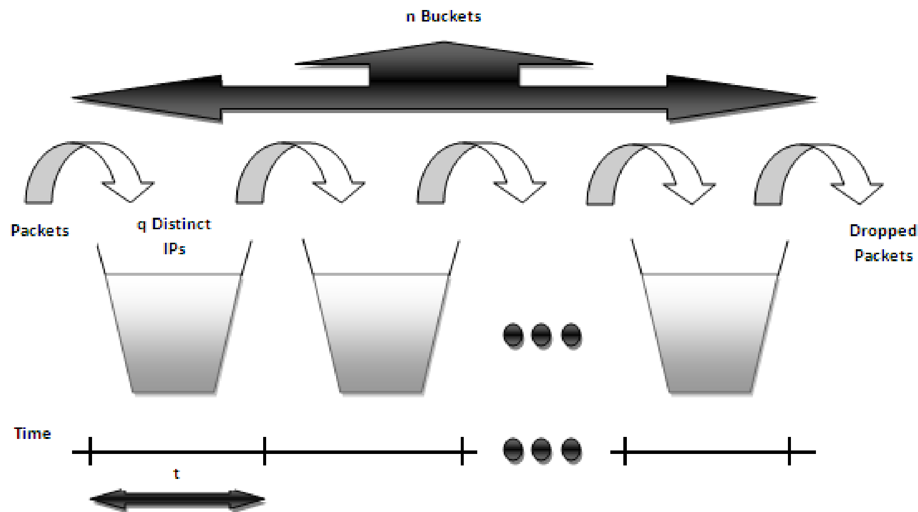

Figure 2-3 Cascading bucket Rate limiting scheme (Wong et al., 2005)

When a bucket is filled with *q* distinct IP addresses, new connection requests are placed into the subsequent bucket, thus each bucket cascades into the next one. Requests in the *i*-th bucket are delayed until the beginning of the *i*+1 time interval. The *n*-th bucket, the last in line, has no overflow bucket and once it is full, new TCP SYN packets without DNS translations are simply dropped. At the end of the *n*t* time periods, another n buckets are reinstated for the next *n*t* time period. This algorithm permits a maximum of *q* distinct IP addresses (without DNS translations) per time interval t and packets (if not dropped) are delayed at most *n*t*.

This scheme can be implemented at the host level or at the edge router of a network. A host-level implementation requires keeping DNS-related statistics on each host. An edge-router-based implementation would require the border router to keep a shadow DNS cache for the entire network.

An attacker can attempt to circumvent the DNS rate limiting mechanism in a number of ways: First, a worm could use reverse DNS-lookups (PTR lookups) to "pretend" that it has received a DNS translation for a destination IP. Jung et al. (Jung et al., 2002) characterizes that PTR lookups are primarily for incoming TCP connections or lookups related to reverse blacklist services. These types of lookups can be easily filtered and not considered as valid entries in the DNS cache. In addition, a PTR lookup prior to an infection attempt will significantly reduce the infection speed. Second, an attacker could setup a fake external DNS server and issue a DNS query for each IP. This threat can be alleviated by establishing a "white-list" of legitimate external DNS servers. Also, the attacker needs a server with a substantial bandwidth to accommodate the scan speed, which is not trivial.

One limitation of the DNS-based rate limiting scheme proposed by Wong et al. is that it looks only for TCP datagrams as the connection initiation and does not consider UDP based traffic. If a worm were to use UDP (such as SQL slammer), the DNS-based rate liming as set out will not be effective.

2.3.2 Automatic Signature Generation (ASG) solutions

Automatic Signature Generation (ASG) solutions refer to approaches which filter incoming traffic to a network and generate signatures on detecting anomalous activity (such as a network worm). For example; Autograph (Kim and Karp, 2004), Earlybird (Singh et al., 2004), Polymorphic Worm Detection Using Structural Information of Executables (Kruegel et al., 2005), PAYL (Wang, Cretu and Stolfo, 2005), PolyS (Paul and Mishra, 2013), LESG (Wang et al., 2010), An Automated Signature Generation Method for Zero-day Polymorphic Worms Based on Multilayer Perceptron Model (Mohammed et al., 2013) etc.

2.3.2.1 Autograph: Towards Automated, Distributed Worm Signature Detection

Kim and Karp (Kim and Karp, 2004) proposed a system which they named Autograph, that automatically generates signatures for novel Internet worms that propagate using TCP transport. Autograph generates signatures by

analysing the prevalence of portions of flow payloads, and thus uses no knowledge of protocol semantics above the TCP level. It is designed to produce signatures that exhibit high sensitivity (high true positives) and high specificity (low false positives). Kim et al. extend Autograph to share port scan reports among distributed monitor instances, and using trace-driven simulation, demonstrate the value of this technique in speeding the generation of signatures for novel worms. Their results elucidate the fundamental trade-off between early generation of signatures for novel worms and the specificity of these generated signatures.

Autograph automatically, without foreknowledge of a worm's payload or time of introduction, detects the signature of any worm that propagates by randomly scanning IP addresses. Kim and Karp assumed that the system monitors all inbound network traffic at an edge network's DMZ. Autograph consists of three interconnected modules: a flow classifier, a content-based signature generator, and tattler- a protocol through which multiple distributed Autograph monitors may share information, in the interest of speeding detection of a signature that matches a newly released worm. Figure 2.4 shows the architecture of the autograph monitor as proposed by Kim and Karp.

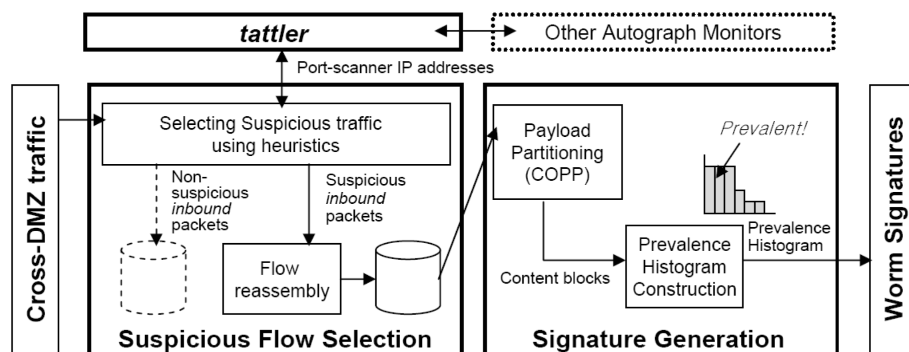

Figure 2-4 Architecture of an Autograph Monitor (Kim and Karp, 2004)

The input of a single Autograph monitor is all traffic crossing the DMZ of an edge network, and its output is a list of worm signatures. There are two main stages in a single Autograph monitor's analysis of traffic. First, a suspicious flow selection stage uses heuristics to classify inbound TCP flows as either suspicious or non-suspicious. After classification, packets for these inbound

flows are stored on disk in a suspicious flow pool and non-suspicious flow pool, respectively. Further processing occurs only on payloads in the suspicious flow pool. Thus, flow classification reduces the volume of traffic that must be processed subsequently. Kim et al. use a simple port-scanner detection technique as a heuristic to identify malicious traffic; they classify all flows from port-scanning sources as suspicious.

Autograph stores the source and destination addresses of each inbound unsuccessful TCP connection it observes. Once an external host has made unsuccessful connection attempts to more than s internal IP addresses, the flow classifier considers it to be a scanner. All successful connections from an IP address flagged as a scanner are classified as suspicious, and their inbound packets written to the suspicious flow pool, until that IP address is removed after a timeout (24 hours in the current prototype). Autograph next selects the most frequently occurring byte sequences across the flows in the suspicious flow pool as signatures. To do so, it divides each suspicious flow into smaller content blocks, and counts the number of suspicious flows in which each content block occurs. Kim and Karp term this count a content block's prevalence, and rank content blocks from most to least prevalent. The intuition behind this ranking is that a worm's payload appears increasingly frequently as that worm spreads. When all worm flows contain a common, worm-specific byte sequence, that byte sequence will be observed in many suspicious flows, and so will be highly ranked. The content block with the greatest prevalence is chosen as signature.

The following are some of the limitations of this approach:

- ***Overload:*** Autograph reassembles suspicious TCP flows. Flow reassembly is costly in state in comparison with processing packets individually, but defeats the subterfuge of fragmenting a worm's payload across many small packets. If Autograph tries to reassemble every incoming suspicious flow, it may be susceptible to a DoS attack.
- ***Source-address-spoofed port scans:*** Source spoofed port scans can be used to mount different attacks, more specific to Autograph: the Tattler

mechanism must carry report traffic proportional to the number of port scanners. An attacker could attempt to saturate tattler's bandwidth limit with spoofed scanner source addresses, and thus render tattler useless in disseminating addresses of true port scanners. A source-spoofing attacker could also cause a remote source's traffic to be included by Autograph in signature generation.

- ***Hit-list Scanning:*** If a worm propagates using a hit-list, rather than by scanning IP addresses that may or may not correspond to listening servers, Autograph's port-scan-based suspicious flow classifier will fail utterly to include that worm's payloads in signature generation.

2.3.2.2 Automated Worm Fingerprinting using Earlybird

Singh et al. (Singh et al., 2004) proposed an automated worm fingerprinting mechanism named Earlybird which detects previously unknown worms and viruses based on two key behavioural characteristics: a common exploit sequence together with a range of unique sources generating infections and destinations being targeted. Singh et al. named their detection approach as content sifting as it automatically generates precise signatures that can then be used to filter or moderate the spread of the worms in the network. Content sifting, is based on two observations: first, that some portion of the content in existing worms is invariant- typically the code exploiting a latent host vulnerability - and second, that the spreading dynamics of a worm are atypical of Internet applications. Simply stated, it is rare to observe the same string recurring within packets sent from many sources to many destinations. By sifting through network traffic for content strings that are both frequently repeated and widely dispersed, we can automatically identify new worms and their precise signatures.

The Earlybird system consists of two major components: sensors and an aggregator. Each sensor sifts through traffic on configurable address space "zones" of responsibility and reports anomalous signatures. The aggregator coordinates real-time updates from the sensors, coalesces related signatures, activates any network-level or host level blocking services and is responsible for administrative reporting and control. Earlybird is implemented in C programming language and the aggregator also uses the MySQL database to

log all events, the rrd-tools library for graphical reporting, and PHP scripting for administrative control. Finally, in order to automatically block outbreaks, the Earlybird system automatically generates and deploys precise content-based signatures formatted for the Snort inline intrusion prevention system.

The prototype of an Earlybird sensor was executed on a 1.6 GHZ AMD Opteron server configured with a standard Linux 2.6 kernel. The server was equipped with two Broadcom Gigabit copper network interfaces for data capture. The Earlybird sensor itself is a single threaded application which executes at user-level and captures packets using the libpcap library.

The Earlybird system has the following limitations:

- If content sifting were to be widely deployed this could create an incentive for worm writers to design worms with little or no invariant content. For example, polymorphic worms encrypt their content in each generation and so-called "metamorphic worms" have even demonstrated the ability to mutate their entire instruction sequence with semantically equivalent, but textually distinct code.
- As an attacker may attempt to evade content sifting algorithms by creating metamorphic worms, he may also attempt to evade Earlybird monitoring through traditional IDS evasion techniques.

2.3.2.3 Polymorphic Worm Detection using Structural Information of Executables

Kruegel et al. (Kruegel et al., 2005) proposed a worm detection technique that detects polymorphic worms. A polymorphic worm is one that mutates as it spreads across the network. This detector technique is based on the structural analysis of binary code that allows one to identify structural similarities between different worm mutations. The approach is based on the analysis of a worm's control flow graph and introduces an original graph colouring technique that supports a more precise characterization of the worm's structure. The technique has been used as a basis to implement a worm detection system that is resilient to many of the mechanisms used to evade approaches based on instruction sequences only.

Polymorphic worms are able to change their binary representation as part of the spreading process. This can be achieved by using self-encryption mechanisms or semantics-preserving code manipulation techniques. As a consequence, copies of a polymorphic worm might no longer share a common invariant substring of sufficient length as observed by Singh et al. (Singh et al., 2004)and the existing systems will not recognize the network streams containing the worm copies as the manifestation of a worm outbreak.

Kruegel et al. observed the fact that some parts of worms contain executable machine code. While it is also possible that certain regions of the code are encrypted, others have to be directly executable by the processor of the victim host (e.g. there will be a decryption routine to decrypt the rest of the worm). Based on this assumption, Kruegel et al. analyze network flows for the presence of executable code. If a network flow contains no executable code, they discard it immediately. Otherwise, they derive a set of fingerprints for the executable regions by using control flow graph extraction and graph coloring techniques.

The worm detection technique presented by Kruegel et al. has the following limitations:

- Firstly, worms that do not use executable code (e.g. worms written in non-compiled scripting languages, for example, Net-Worm: W32/Santy.A (F-Secure: Net-Worm:W32/Santy.A, 2004), written in Perl,JS.Gigger.A@mm (Symantec: JS.Gigger.A@mm, 2002), written in JavaScript) will not be detected by their worm detection system.
- Secondly, the proposed prototype of Kruegel et al. operates on offline data. But this technique has one distinct advantage over Autograph (Kim and Karp, 2004) and Earlybird (Singh et al., 2004) as it detects polymorphic worms which other techniques are not designed to detect.

2.3.2.4 Anomalous Payload-based Worm Detection and Signature Generation

Wang et al. (Wang, Cretu and Stolfo, 2005) proposed a worm detection system, which they named PAYL, for the detection of zero-day worms. The principle behind PAYL is that a new zero-day attack will have content data never before seen by the victim host, and will likely appear quite different from normal data and be deemed anomalous. The approach proposed by Wang et al. is based on ingress/egress anomalous payload correlation, and uses no scan or probe information. The key idea is that a newly infected host will begin sending outbound traffic that is substantially similar (if not exactly the same) as the original content that attacked the victim (even if it is fragmented differently across multiple packets). Correlating ingress/egress anomalous payload alerts can detect worm propagation and stop the worm spread from the very moment it first attempts to propagate itself, instead of waiting until the volume of outgoing scans suggests full-blown propagation attempts.

Although PAYL is a fully automatic, “hands-free” online anomaly detection sensor system but it has following limitations:

- PAYL is not a real time system and is based on analysing network traces.
- The range of worms tested by Wang et al. is limited in number and scope.

2.3.2.5 PolyS: Network-based Signature Generation for Zero-day Polymorphic Worms

Paul and Mishra (Paul and Mishra, 2013) proposed PolyS, a network based automated signature generation scheme to thwart zero-day polymorphic worms. They presented a novel architecture for successfully matching a polymorphic worm payload that reduces the noise in the suspicious traffic pool, thus enhancing the accuracy of worm’s signature and a signature generation algorithm for successfully matching polymorphic worm payload with higher speed and memory efficiency.

2.3.2.6 LESG: Thwarting Zero-Day Polymorphic Worms With Network-Level Length-Based Signature Generation

Wang et al. (Wang et al., 2010) proposed network-based length-based signature generator (LESG) for generating vulnerability-driven signatures for buffer overflow worms at the network level without any host-level analysis of worm execution or vulnerable programs. This is the first attempt to generate vulnerability-driven signatures at network level. They build a field hierarchy model, and formally define the length based signature generation problem based on it. The proposed algorithm designed to solve that problem has good accuracy even under deliberate noise injection attacks. Wang et al. evaluated LESG against real-world vulnerabilities of various protocols and real network traffic and demonstrated that LESG is fast, noise tolerant and has efficient signature matching.

2.3.2.7 An Automated Signature Generation Method for Zero-day Polymorphic Worms Based on Multilayer Perceptron Model

Mohammed et al. (Mohammed et al., 2013) proposed a signature generation system for zero-day polymorphic worms based on the Double-honeynet system, k-means clustering algorithm and a Multilayer Perceptron Model. The Double-honeynet system is used to collect polymorphic worm samples as a first step, while the second step is the signature generation for the collected samples by using a k-means clustering algorithm and a Multilayer Perceptron Model. The k-means clustering algorithm separates different types of collected polymorphic worms into different clusters. The Multilayer Perceptron Model then generates signatures for each cluster.

2.3.2.8 Automated Signature Generation for Zero-day Polymorphic Worms Using a Double-honeynet

Mohssen M. Z. E. Mohammed (Mohammed, 2012) designed a system of automated signature generation for zero-day polymorphic worms using a double-honeynet, Modified Knuth-Morris-Pratt (MKMP) algorithm and a Modified Principal Component Analysis (MPCA) algorithm. The polymorphic worm instances are collected by designing a novel double honeynet system, that allows unlimited honeynet outbound connections to collect all polymorphic worm instances. Then, a Modified Knuth-Morris-Pratt (MKMP) Algorithm, which is string matching based, and a Modified Principal

Component Analysis (MPCA), which is statistics based, are used to generate the signatures. The MKMP algorithm compares the polymorphic worms substrings to find the multiple invariant substrings that are shared between all polymorphic worm instances and uses them as signatures, where as the MPCA determines the most significant substrings that are shared between polymorphic worm instances and use them as signatures.

2.3.2.9 Efficient Hybrid Technique for Detecting Zero-Day Polymorphic Worms

Ratinder Kaur and Maninder Singh (Kaur and Singh, 2014) presented a technique for detecting zero-day polymorphic worms, which is based on both signature detection and anomaly detection techniques. Honeynet is used as an anomaly detector to identify and capture new attacks. After detection, the new attacks are validated for polymorphism and finally signatures are generated for discovered zero-day polymorphic worms to assist in containing them.

2.3.3 Behavior Based Signature Detection (BSD) solutions

Behaviour Signature Detection (BSD) solutions refer to approaches which look for anomalous behaviour signatures in network traffic. A behavioural signature describes aspects of behaviour of a particular worm that are common across the manifestations of a given worm and that span its nodes in temporal order. Characteristic patterns of worm behaviour in network traffic include (Ellis et al., 2004), (Whyte, Kranakis and Oorschot, 2005):

- Sending similar data from one host to the next
- Tree-like propagation and reconnaissance
- Changing a server into a client
- Lack of DNS lookup
- Lack of ARP lookup

Various Behaviour Signature Detection (BSD) Solutions are: Network Application Architecture (NAA) (Ellis et al., 2004), DNS based detection of Scanning Worms (Whyte, Kranakis and Oorschot, 2005) and ARP based detection of worms (Whyte, van Oorschot and Kranakis, 2005).

2.3.3.1 Network Application Architecture (NAA) - A Behavioral Approach for Worm Detection

Ellis et al. (Ellis et al., 2004)proposed a worm detection approach which they term Network Application Architecture (NAA) and employs behavioural signatures to detect worms. A behavioural signature describes aspects of behaviour of any particular worm that are common across the manifestations of a given worm and that span its nodes in temporal order. Characteristic patterns of worm behaviour in network traffic include:(1) sending similar data from one host to the next, (2) tree-like propagation and reconnaissance, and (3) changing a server into a client.

The approach presented by Ellis et al. differs from those used in contemporary enterprise postures in two ways. The first characteristic of contemporary postures is the reliance on a particular type of signature-based intrusion detection. In the contemporary case, a signature is a regular expression known a priori. Most signatures deployed in current intrusion detection systems (IDSs) focus on detecting specific regular expressions in network packets. The use of a previously unknown version of an exploit will evade detection. The behavioural detection approach contrasts from this form of signature-based detection. Instead of looking for fixed regular expressions in payloads, the behavioural approach focuses on detecting patterns at a higher level of abstraction. Ideally, the patterns are inherent behaviours of worm spread and distinct from normal network traffic. The frequency of and interrelationships between behaviours improve detection accuracy. To evade a behavioural signature requires a change in fundamental behaviour, not just its network footprint. Modifying behaviours to evade detection may be much more challenging.

Ellis et al. presented three behavioural signatures. The first is that the inputs and outputs of a host are related for all non-discriminating worms that do not have a polymorphic network footprint. The second is that non-discriminating worms turn servers of a service into clients of the service. Together, these two signatures identify behaviour, which, on a per-host basis indicates a change in logic. The third signature is identifying a tree-like structure in communication

patterns emerging from infected nodes. As the worm spreads, infected hosts contact other hosts. The resulting tree-like communications have features in common, possibly including the previous two signatures. NAA impacts the sensitivity of this behavioural approach. That is, the distribution of hosts and network applications across those hosts impacts the normal traffic patterns on an enterprise network. Under certain NAAs, constraints are placed on traffic patterns, which worm traffic patterns violate. Violations of these constraints are straightforward to detect and hence provides the proof of worm activity.

The most significant advantage of NAA is its ability to detect classes of worms without a priori information on any specific worm by employing behavioural signatures. However, the NAA approach will not be effective in case of fast spreading worms as it lacks the functionality of spreading malware warnings to uninfected sites in a timely manner.

2.3.3.2 DNS-based Detection of Scanning Worms in an Enterprise Network

Whyte et al. (Whyte, Kranakis and Oorschot, 2005) proposed DNS-based detection of scanning worms in an enterprise network. DNS-based detection relies on the correlation of Domain Name Service (DNS) queries with outgoing connections from an enterprise network. Whyte et al. claim the following improvements over existing scanning worm detection techniques: (1) the possibility to detect worm propagation after only a single infection attempt; (2) the capacity to detect zero-day worms; and (3) a low false positive rate.

Whyte et al. divided the enterprise network into segments called cells. Each cell contains a worm containment host to confine and contain worm infection. Whyte et al. define a cell as all hosts within the same subnet serviced by a distinct root DNS server. The propagation of fast-scanning worms can be characterized as: local to local (L2L), local to remote (L2R), or remote to local (R2L). In L2L propagation, a scanning worm targets hosts within the boundaries (subnets) of the enterprise network. Topological scanning worms employ this strategy. L2R propagation refers to a scanning worm within an enterprise network targeting hosts outside of the network boundary. Finally,

R2L propagation refers to worm scanning from the Internet into an enterprise network. The DNS-based worm propagation detection method proposed by Whyte et al. detects L2R worm propagation and worm propagation between local cells.

Figure 2.5 shows an example of an operational prototype of the DNS-anomaly based detection system.

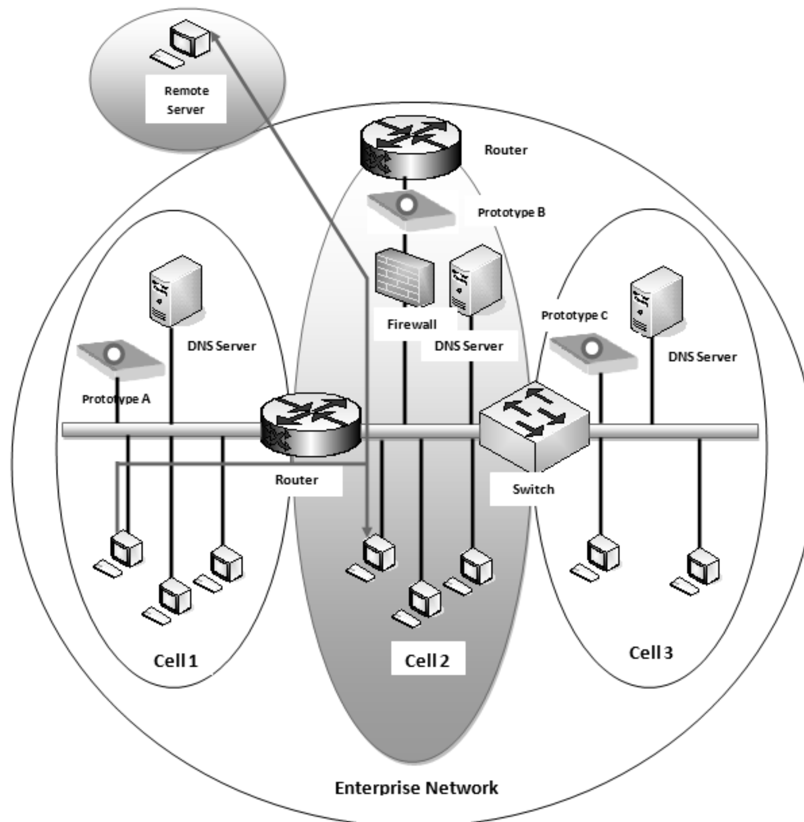

Figure 2-5 DNS Anomaly-based Detection Deployment (Whyte, Kranakis and Oorschot, 2005)

Prototype A in cell 1 monitors activity between cell 1 and cell 2. Cell 2 contains the sole ingress/egress point for the enterprise network. Prototype B, from its vantage point in cell 2, monitors activity from all cells within the enterprise network to external hosts. Finally prototype C monitors activity between cell 3 and cell 2. In the case that a host in cell1isinfected with a scanning worm, the infected host will begin scanning to locate susceptible hosts both within cell 2 and the Internet. The prototype host in cell 1 will

detect the scanning activity to cell 2 and generate an alert. The prototype host in cell 2, at the enterprise gateway, will detect scanning activity from cell 1 to the Internet and generate an alert.

The software system design presented by Whyte et al. uses the libpcap library (TCPDUMP & LIBPCAP, 2008) and is comprised of two logical components: the PPE and DCE. The Packet Processing Engine (PPE) is responsible for extracting the relevant features from the live network activity or saved network traces. The DNS correlation engine (DCE) maintains in state all relevant DNS information, a white-list (which contains applications which do not rely on DNS lookup), and numeric IP addresses embedded in HTTP packets extracted by the PPE. This information is used to verify both outgoing TCP connections and UDP datagrams. In this context, verifying means ensuring that the destination IP address of an outgoing TCP connection or UDP datagram can be attributed to a DNS query, an HTTP packet, or an entry in the whitelist. The software can process either live network traffic or saved network traces in pcap file format. To detect L2R worm propagation, the software system must be deployed at all external network egress/ingress points. To detect worm propagation between network cells, a system would need to be deployed in each cell at the internal ingress/egress points (see Figure 2.5).

This detection approach has two limitations as it cannot detect intra-cell and Internet to enterprise (R2L) worm propagation.

2.3.3.3 ARP-based Detection of Worms within an Enterprise Network

Whyte et al. (Whyte, van Oorschot and Kranakis, 2005) proposed another anomaly based worm detection technique that protects internal networks from scanning worm infections. Implemented in software, this detection approach relies on an aggregate anomaly score, derived from the correlation of Address Resolution Protocol (ARP) activity from individual network attached hosts. Whyte et al. divided the network into cells and seek to detect scanning worm activity within cells. According to the authors, the scanning worm targeting hosts within its own network cell exhibits anomalous behaviour distinct from normal ARP activity; an infected host generates unusual ARP request activity as it tries to infect susceptible hosts within its respective network cell. More

specifically, intra-cell worm initiated scans result in discernible behavioural changes in the amount and pattern of ARP request activity of the infected hosts, because a scanning worm targeting same-cell hosts triggers the broadcast of anomalous ARP “who has” requests.

The ARP-based technique proposed by Whyte et al. is based on the following three factors; from them they derive an anomaly score for each individual host and use this as an infection indicator for each host within a cell:

- ***Peer list:*** connections to hosts outside the set of internal hosts, a host normally interacts with.
- ***ARP activity:*** increases in the average number of ARP requests each host issues per unit time.
- ***Internal network dark space connections to vacant internal IP addresses (i.e. addresses not bound to any active hosts)***: The greater the anomaly score attributed to a network host, the more likely it is infected with a scanning worm.

Figure 2.6 shows an enterprise network divided into cell structures. Hosts that reside within the same network cell use ARP rather than the Domain Name Service (DNS) to communicate.

The scheme proposed by Whyte et al. provides a novel approach for worm detection but it has following limitations:

- It cannot detect R2L and L2R remote propagation. This technique is probably therefore best suited to be used in combination with another technique which covers R2L and L2R.

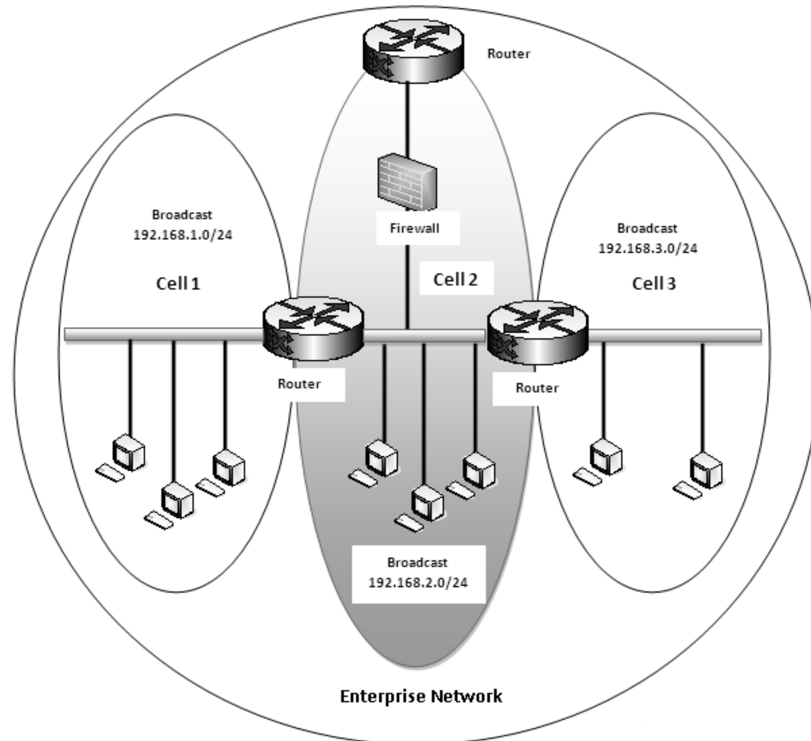

Figure 2-6 ARP-based Detection of Worms within an Enterprise Network (Whyte, van Oorschot and Kranakis, 2005)

2.3.4 Leap Ahead (LA) solutions

Leap Ahead (LA) solutions, seek to spread malware warnings to network segments not yet affected, and thus potentially stop the worm from reaching its full saturation potential. These strategies share cooperative information either hierarchically or using peer to peer based models. For example Cooperative Alert Sharing Scheme Using a “Friends protocol” (Nojiri, Rowe and Levitt, 2003), COVERAGE (Anagnostakis et al., 2003), Very Fast Containment of Scanning Worms (Weaver, Staniford and Paxson, 2004), Monitoring and Early Warning on Internet Worms (Zou et al., 2003).

2.3.4.1 Cooperative Response Strategies for Large Scale Attack Mitigation

Nojiri et al. (Nojiri, Rowe and Levitt, 2003) proposed a cooperative alert sharing scheme using a “Friends protocol” under which each node (domain gateway) pre-selects a set of friends with which to share worm detection indicators, and is also selected by other domains gateways to receive reports. Although this technique provides an effective way of sharing worm warnings,

it is ineffective in the case of slow spreading worms and also lacks a good worm detection mechanism.

2.3.4.2 COVERAGE

Anaganostaki et al. (Anagnostakis et al., 2003) proposed a variation of the LA scheme called COVERAGE, in which a node randomly selects a set of remote nodes to poll for worm reports at periodic intervals. The LA concept is effective in spreading malware warnings to uninfected network segments, but these solutions are limited in terms of their implementation in current networks.

2.3.4.3 Very Fast Containment of Scanning Worms

Weaver et al. (Weaver, Staniford and Paxson, 2004) proposed a worm detection technique by devising mechanisms for cooperation that enable multiple containment hosts to more effectively detect and respond to an emerging infection. A key problem in containment of scanning worms is efficiently detecting and suppressing the scanning. Since containment blocks suspicious hosts, it is critical that the false positive rate be very low. Additionally, since a successful infection could potentially subvert any software protection put on the host, containment is best effected on the network gateway rather than on end-hosts. Weaver et al. developed a scan detection and suppression algorithm based on a simplification of the Threshold Random Walk (TRW) scan detector.

Weaver et al. augmented the containment system by employing cooperation between the containment hosts that monitor different cells. By introducing communication between these hosts, they can dynamically adjust their thresholds to the level of infection. Weaver et al. showed that introducing a very modest degree of bias that grows with the number of infected cells makes a dramatic difference in the efficacy of containment above the epidemic threshold. Thus, the combination of containment coupled with cooperation holds great promise for protecting enterprise networks against worms that spread by address-scanning.

Weaver et al. implemented the prototype on an ML300 demonstration platform manufactured by Xilinx. This board contains 4 gigabit Ethernet interface, a

small FPGA, and a single bank of DDR-DRAM. The DRAM bank is sufficiently large to meet the design goals; while the DRAM's internal banking should enable both the address and connection tables to be implemented.

Although, Weaver et al. presented a novel approach for worm detection using TRW algorithm, the system lacks enterprise level testing.

2.3.4.4 Monitoring and Early Warning of Internet Worms

Zou et al. (Zou et al., 2003) proposed a novel algorithm for early detection of the presence of a worm and the corresponding monitoring system. Based on an epidemic model and observation data from the monitoring system, by using the idea of "detecting the trend, not the rate" of monitored illegitimate scan traffic, Zou et al. used a Kalman filter to detect a worm's propagation at its early stage in real-time to detect the overall vulnerable population size.

The Kalman filter detects the propagation of a worm in its early stage based on observed illegitimated scan traffic, which includes both real worm scans and background noise. The Kalman filter will not only make use of the correlation of the history trace of observation data (not just a burst of traffic at one time), but also the dynamic trend of the propagation of a worm - at the beginning of a worm's spreading when there are little human counteractions or network congestions, a worm propagates almost exponentially with a constant, positive infection rate. The Kalman filter is activated when the monitoring system encounters a surge of illegitimate scan activities. If the worm infection rate estimated by the Kalman filter stabilizes and oscillates a little bit around a constant positive value, it is claimed that the illegitimate scan activities are mainly caused by a worm, even if the estimated value of the worm's infection rate is still not well converged. If the illegitimate scan traffic is caused by non-worm noise, the traffic will not have the exponential growth trend, and the estimated value of infection rate would oscillate around without a fixed central point, or it would oscillate around zero. In other words, the Kalman filter is used to detect the presence of a worm by detecting the trend, not the rate, of the observed illegitimated scan traffic. In this way, the unpredictable, noisy, illegitimate scan traffic we observe everyday will not cause many false alarms

to the detection system - such background noise will cause significant challenges to traditional threshold-based detection methods.

Figure 2.7 shows a schematic of the monitoring system proposed by Zou et al. with two kinds of monitors:

- ***Ingress Scan Monitors:*** Ingress scan monitors are located on gateways or border routers of local networks. They can be the ingress filters on border routers of local networks or separated passive network monitors. The goal of an ingress scan monitor is to monitor scan traffic coming into a local network by logging incoming traffic to unused IP addresses in the network.
- ***Egress Scan Monitors:*** An egress scan monitor is located at the egress point of a local network. It can be set up as a part of the egress filter on the routers of a local network. The goal of an egress scan monitor is to monitor the outgoing traffic from a network to infer a scan behaviour of a potential worm. Ingress scan monitors listen to the global traffic on the Internet; they are the sensors of global worm incidents (referred to as a "network telescope" in (Moore, 2002)).

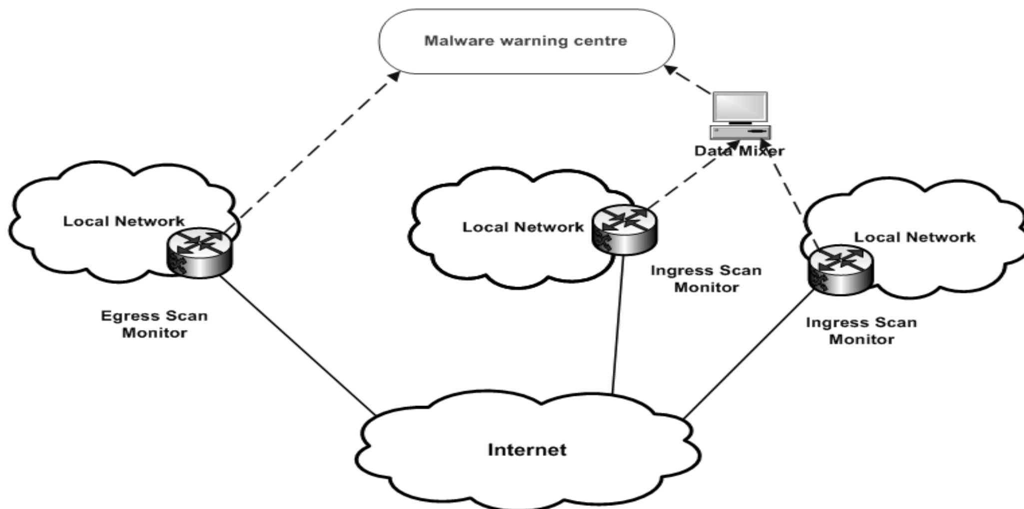

Figure 2-7 Worm Monitoring System (Zou et al., 2003)

In order to achieve early warning of activity in real-time, distributed monitors are required to send observation data to the Malware warning centre (MWC) continuously without significant delay, even when the worm scan traffic has

caused congestion to the Internet. For this reason, a tree-like hierarchy of data mixers can be set up between monitors and the MWC: the MWC is the root; the leaves of the tree are monitors. The monitors close to a data mixer in the network send observed data to the data mixer. After fusing the data together, the data mixer passes the data to a higher level data mixer or directly to MWC. An example of data fusion is the removal of redundant addresses from the list of infected hosts. However, the tree structure of data mixers create single points of failure, thus there is a trade-off in designing this hierarchical structure.

The detection approach of Zou et al. has the following limitation:

- Although this approach provides an idea of setting up a monitoring system with the help of simulations for worm detection, it clearly lacks any enterprise level implementation and testing.

2.3.5 Predesigned-Preventative (PP) solutions

Predesigned-Preventative (PP) solutions are considered to be those approaches which are designed to disrupt the discovery of susceptible nodes within an address space, potentially by dynamically altering the connectivity of networks or end nodes in the presence of worm propagation. For example, Epidemic Profiles and Defense of Scale-Free Networks (Briesemeister, Lincoln and Porras, 2003), Least Effort Strategies for Cyber Security (Gorman et al., 2003), A Virtual Honeypot Framework (Provos, 2004), Honeypot worm detection system “Billy Goat” (Riordan, Zamboni and Duponchel, 2006), Router-based Billy Goat (RBG) (Zamboni, Riordan and Yates, 2007) and Network Address Space Randomization (NASR) (Antonatos et al., 2007).

2.3.5.1 Epidemic Profiles and Defense of Scale-Free Networks

Briesemeister et al. (Briesemeister, Lincoln and Porras, 2003) discussed the idea of percolation theory or, epidemic spread, in artificial scale-free networks to suggest how networks could be designed to delay the spread of propagating malware while still maintaining high reliability of network links.

2.3.5.2 Least Effort Strategies for Cyber Security

Gorman et al. (Gorman et al., 2003) studied the use of scale-free properties within the autonomous system (AS) map of the Internet, and proposed that the

concentration of worm filtering services on the nodes with the highest connection density would yield the greatest return while disrupting the minimum set of network hosts.

2.3.5.3 A Virtual Honeypot Framework

Provos (Provos, 2004) suggested the placement of honeypot hosts in a network that engage in slow connection dialogs as a method to dramatically slow an aggressive worm's ability to discover susceptible hosts within an address space. Provas presented "Honeyd", a framework for virtual honeypots that simulates virtual computer systems at the network level. The simulated computer hosts appear to run on unallocated network addresses. To deceive network fingerprinting tools, Honeyd simulates the networking stack of different operating systems and can provide arbitrary routing topologies and services for an arbitrary number of virtual hosts.

Honeyd, is an effective system to detect worms and spam. Its performance measurements showed that a single 1.1 GHz Pentium III can simulate thousands of virtual honeypots with an aggregate bandwidths of over 30 MBit/s and that it can sustain over two thousand TCP transactions per second. But, it is ineffective in detecting against fast spreading zero-days worms.

2.3.5.4 Building and Deploying Billy Goat, Worm-Detection System

Riordan et al. (Riordan, Zamboni and Duponchel, 2006) proposed a honeypot worm detection system "Billy Goat" which is widely deployed throughout IBM. The deployment within IBM covers the entirety of the corporate intranet, automatically gathering data from approximately 1.2 million virtual sensors, centralizing the data to form a single coherent model of suspicious network activity, and analysing this model for evidence of worm activity. Billy Goat is designed to take advantage of the propagation strategies of worms. To discover hosts to infect, most worms try to connect to IP addresses selected at random or scan entire ranges of addresses. By doing so, they find most of the hosts in a network, but they also try to connect to a large number of unused addresses. The fundamental premise of Billy Goat is responding to traffic directed to unused IP addresses.

Billy Goat is implemented as a specialized Linux distribution, which self-installs on standard PC requiring only basic configuration information. It was the intention to make Billy Goat as appliance-like as possible, so that it can be deployed with minimum effort throughout a large network. Billy Goat includes extensive self-monitoring and recovery mechanisms that monitor host activity and correct or reinitialize errant components, including the host itself (e.g. reboot). Different deployment modes can be used and combined to direct such traffic to Billy Goat.

- Static routes
- ARP Spoofing
- Billy Goat as default LAN route
- ICMP-based Billy Goat

In summary, Billy Goat is an effective approach for detection and prevention of worms in an intranet; it lacks the capability to detect and mitigate fast spreading zero-day worm outbreak on the Internet.

2.3.5.5 Boundary Detection and Containment of Local Worm Infections

Zamboni et al. (Zamboni, Riordan and Yates, 2007) proposed a system for detecting scanning-worm infected hosts in a local network. Infected hosts are detected after a few unsuccessful connection attempts such as by logging ICMP unreachable messages, refused connections and timeouts, and in cooperation with the border router, their traffic is redirected to a honeypot for worm identification and capture.

Zamboni et al. used Router-based Billy Goat (RBG), a specialized worm-detecting honeypot, as the host to which traffic is redirected. RBG is a mechanism that adds dynamic discovery of external unused or unreachable IP addresses and redirects traffic sent to such addresses to a honeypot for processing and response. This dynamic assignment vastly extends the monitoring abilities of the honeypot. The idea of RBG is to trigger traffic redirection upon detection of failed connection attempts. Such attempts can be detected by the following mechanisms:

- Receipt of ICMP unreachable messages

- Timed-out initial connections
- Detection of refused connections

Under normal conditions, when a host tries to contact an unreachable destination or service, one of the three error conditions mentioned above occurs. When using RBG, the error condition is intercepted. For example, in the case of an ICMP error message, the following sequence (illustrated in Figure 2.8) takes place:

- The internal host sends the first packet of the connection.
- The external router sends back an ICMP Unreachable message. The local router intercepts it and automatically generates a rule to route future packets to this unreachable destination, to the honeypot and also sends the original packet to the honeypot.
- The Billy Goat system receives the packet and replies to it, spoofing the destination host. The internal host gets the reply he wanted and will consider the destination host as being up.

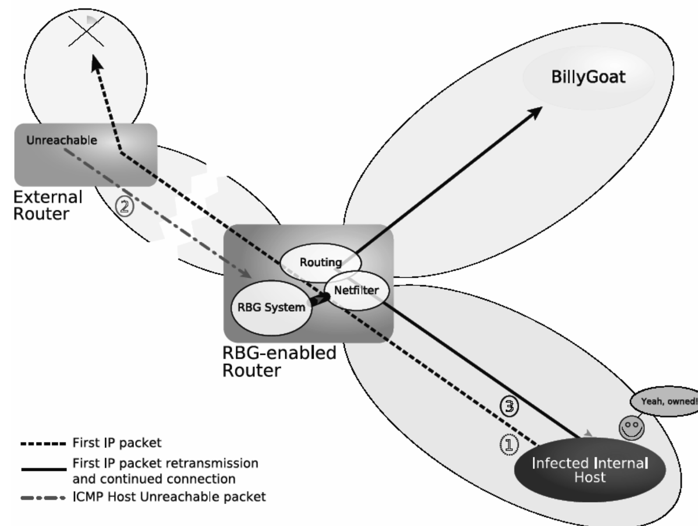

Figure 2-8 The “unreachable destination” Behaviour using the RBG Architecture (Zamboni, Riordan and Yates, 2007)

The ideal place to put the RBG logic and mechanisms is in a border router as described by Zamboni et al. They have used this approach in their implementation using a Linux-based router. However, it would also be

possible to implement RBG as a bridge placed between the border router and the internal network, monitoring traffic and remotely reconfiguring routes. This mode of deployment would make it easier to adopt RBG without modifying deployed routers.

RBG offers the significant benefit of detecting local infections locally, providing a valuable tool to network administrators, and it helps perform local containment of worm infections, thereby preventing unwanted traffic from leaving the local network. But it has following limitations:

- ***Detection of scanning worms only:*** By design, RBG will only detect and redirect traffic produced by hosts that are scanning non-existent IP addresses. Hit-list worms, and other types of malware that direct their attacks against existing hosts and services will not be detected by RBG.
- ***IP spoofing:*** Using IP address spoofing, an attacker inside the local network could abuse RBG and make it isolate a local IP address from the outside, using the source flooding detection feature of RBG. This attack may be mitigated using MAC address checking and filtering.

2.3.5.6 Defending against Hit-list Worms using Network Address Space Randomization

Antonatos et al. (Antonatos et al., 2007) proposed a proactive worm defense mechanism called Network Address Space Randomization (NASR) whose objective is to harden networks specifically against hit-list worms. The idea behind NASR is that hit-list information could be rendered stale very rapidly if nodes are forced to frequently change their IP addresses on a regular basis. NASR limits or slows down hit-list worms and forces them to exhibit features that make them easier to contain at the perimeter.

A basic form of NASR can be implemented by configuring the DHCP server to expire DHCP leases at intervals suitable for effective randomization. The DHCP server would normally allow a host to renew the lease if the host issues a request before the lease expires. Thus, forcing address changes even when a host requests to renew the lease before it expires requires some minor modifications to the DHCP server. Fortunately, it does not require any

modifications to the protocol or the client. Antonatos et al. have implemented an advanced NASR-enabled DHCP server, called Wuke-DHCP, based on the ISC open-source DHCP implementation. To minimize the “collateral damage” caused by address changes, Antonatos et al. introduced two modules in their DHCP implementation: an activity monitoring module, and a service fingerprinting module.

In the prototype implementation, Antonatos et al. used three timers on the DHCP server for controlling host addresses. The refresh timer determines the duration of the lease communicated to the client. The client is forced to query the server when the timer expires. The server may or may not decide to renew the lease using the same address. The soft-change timer is used internally by the server to specify the interval between address changes, assuming that the flow monitor does not report any activity for the host. A third, hard-change timer is used to specify the maximum time that a host is allowed to keep the same address. If this timer expires, the host is forced to change address, despite the disruption that may be caused.

Antonatos et al. proposed a novel system for worm prevention but it has some practical constraints as described below:

- It is not feasible to change the IP address of servers like Domain Name Server (DNS), Web Servers as public DNS servers require a considerable amount of time to replicate on the Internet.
- Many applications are not designed to tolerate connection failures. For instance, NFS clients often hang when the server is lost, and do not transparently re-resolve the NFS server address from DNS before reconnecting.

2.3.6 Mobile Combat (MC) solutions

Mobile Combat (MC) solutions refer to approaches which involve an active strategy of interception and rapid patching. These techniques eliminate propagating malware by distributing a mobile self-replicating code module that searches out for signs of a malicious resident code and vaccinates infected hosts through patching or some other removal method. For example, Predators:

Good Mobile Code Combat against Computer Viruses (Toyoizumi and Kara, 2002), Models of Active Worm Defense (Nicol and Liljenstam, 2005) and Mobile combat / Beneficial worms "in the wild" (Symantec: W32.Welchia.Worm, 2003), (Weaver et al., 2003) etc .

2.3.6.1 Predators: Good Mobile Code Combat against Computer Viruses

Toyozumi and Kara (Toyoizumi and Kara, 2002) presented an analysis of a predatory vaccination application called Predator. They employed the biologically inspired Lotka-Volterra equation (Lotka, 1925), (Volterra, 1926) to model the interaction of the predator-prey relationship between the malicious code and mobile predator vaccination, with the goal of minimizing the number of predators required to eliminate the malware threat. Their paper proposed that a small number of good predators, of the order of a few thousand, could contain an aggressive large-scale worm such as Code-Red.

2.3.6.2 Models of Active Worm Defense

Nicol and Liljenstam (Nicol and Liljenstam, 2005) investigated different active defense propagation models, from simple scanning systems that race against worms to patch susceptible hosts, to sniper worms that behave in a similar way to the Predator model. Using a discrete stochastic model, the author proved that these approaches can be strongly ordered in terms of their worm fighting capability. Using a continuous model, Nicoland and Liljenstam consider effectiveness in terms of the number of hosts that are protected from infection, the total network bandwidth consumed by the worms and the defences, and the peak scanning rate the network endures while the worms and defences battle.

2.3.6.3 Mobile Combat /Beneficial worms "In the Wild"

The following are some examples of mobile combat or beneficial worms which have been implemented and released in order to combat against harmful worms:

- ***Welchia:*** It is Blaster worm variant (Symantec: W32.Welchia.Worm, 2003), released to mitigate the spread of the Blaster worm. It exploited the same vulnerability at the same TCP port as Blaster to propagate and immunized a susceptible host by exploiting the vulnerability and downloading the MS03-026 patch then rebooting. But, the Welchia

worm was unsuccessful in achieving its goals of stopping Blaster due to fact that it utilized massive bandwidth on the Internet by downloading patches from the vendor server (windowsupdate.com), thereby, launching a denial of service attack at windowsupdate.com.

- ***CRClean:*** CRClean is a Code Red II variant (Weaver et al., 2003) which exploits a buffer overflow vulnerability in the index server plug-in in Microsoft IIS Server. It only spreads to hosts that have attempted to attack it, referred to as passive scanning. It silently runs on a host, waiting and listening for a Code Red attack. When CRClean intercepts an attack scan from Code Red infected hosts, it launched a counter attack at the host that has launched the attack, removes Code Red and installs CRClean. CRClean was never released on the Internet.

Although MC solutions present an effective approaches for worm detection and patching, these approaches are not effective in terms of fast spreading worms like Slammer. Secondly, legality of such solutions will be a big issue as it is illegal and unethical to launch a worm even for constructive purpose.

2.3.7 Hybrid Quarantine Defense (HQD) solutions

Hybrid Quarantine Defense (HQD) solutions use a combination of different worm detection and prevention solutions. For example, A Hybrid Quarantine Defense (Porras et al., 2004) uses combination of RL and LA solutions.

2.3.7.1 A Hybrid Quarantine Defense

Porras et al. (Porras et al., 2004) proposed a hybrid quarantine defense system for worm detection and prevention by combining rate limiting mechanisms and a leap-ahead solution using the friends quarantine strategy. The resource limitation strategy proposed by Porras et al. focuses on limiting the number of outbound nodes that an internal host may contact per unit time. This strategy is motivated from the observation that during normal operation, the rate of outbound connections to unique hosts is relatively small, and that rate generally increases when a host is infected by a scan based worm in proportion to the aggressiveness with which the worm seeks susceptible nodes.

Figure 2.9 illustrates the connection rate-limiting algorithm, as implemented by Porras et al. in their simulation. Rate limiting is performed at the gateway of each domain, rather than at the individual internal node. Each internal node is allowed to make ≤ N outbound connections per time unit. Outbound connections beyond N per unit time are dropped by the gateway. A host can make any number of internal connections without interference, and thus once a worm enters the domain, it may spread to all internal nodes without interference. A threshold limit of N = 10 addresses per unit time is selected as the default parameter for this algorithm.

For their leap-ahead strategy, Porras et al. implemented a variation of the Friends algorithm (Zou et al., 2003). Essentially, each domain head (gateway) m ∈ M selects F = G − 1 friends. This selection defines group size G over the population M. The group memberships of one domain head overlap so that one domain head is a member of multiple groups, in which the other domain head selected this one as a friend. Under the Friends protocol, each gateway activates port or content-based filtering, when it receives enough alerts from friends (including itself) to indicate the presence of a worm. No single alert is sufficient to trigger filtering, and thus Friends gateways tolerate an adjustable amount of false alarms before they must react to an emerging worm threat. The warning state proceeds to temporally decay until it drops into a state in which filtering is removed from the gateway, but may be raised indefinitely while worm activity indicators persist.

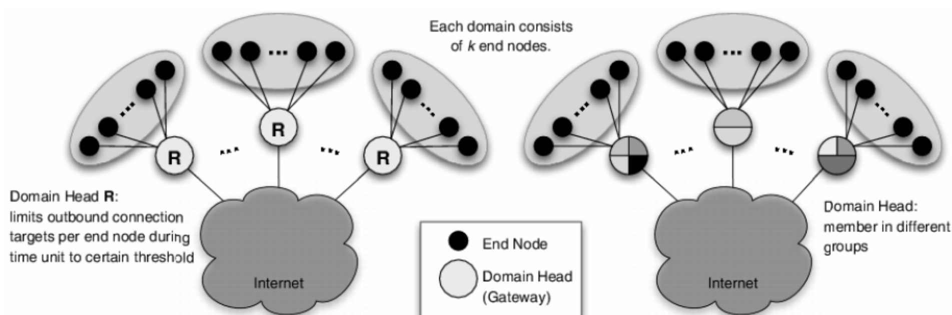

Figure 2-9 Connection Rate Limitations and Friends Overview (Porras et al., 2004)

In the combined defense strategy proposed by Porras et al., each gateway will implement a connection rate limiting defense in parallel with the Friends protocol. The objective is to employ each rate limiter to effectively slow down the propagation of aggressive worms, allowing Friends messages to propagate to groups and activate a defensive posture in time to halt infection growth before full saturation is reached. The triggering of node rate limiting can itself act as one indicator of worm activity, and extensions of this overlay solution could include feedback loops in which the rate-limiting threshold maybe adjusted by the accumulation of Friends messages at predefined thresholds.

The Hybrid strategy proposed by Porras et al. yields substantial performance improvements, beyond what either technique provides independently but the resource limiting technique is prone to high rate of false positives due to the rate limiting algorithm.

2.3.8 Defensive Worms (DW) solutions

Defensive Worms (DW) solutions employ defensive worms to combat against malicious worms. A defensive worm (Ziyad, 2011) refers to a controlled, self-propagating, and self-contained network program that when released does not violate the laws issued by a legislative body and whose purpose of release is beneficial. Ziyad AL-Salloum (Ziyad, 2011) proposed two defensive worms Seawave I and Seawave II.

- ***Seawave I:*** Seawave I is a novel controlled, topology-aware, interactive, self-replicating, self-propagating, and self-contained network vulnerability mitigation system (or vulnerability mitigation worm), that utilizes CAM and STP information to propagate.
- ***Seawave II:*** Seawave II is based on STP, CAM, ARP, and OSPF, in which they enhanced and improved the defensive worm by adding edge node failure recovery, network backbone traversal, and intermittent node detection and recovery.

Both these approaches were simulated on NS2 with 100 to 8000 nodes on a LAN, an approach which in isolation, clearly lacks the real time testing. Secondly, Seawave I and Seawave II do not address the mitigation technique

on wide area networks i.e. the Internet, thus making them impractical to deploy in case of a fast spreading zero-day worm.

2.4 Worm Testing Environments

Various network and malware testing environments have been built and proposed in the past which can be classified into the following categories:

- Physical network testbeds
- Simulation testbeds
- Emulation testbeds
- Full system virtualization testbeds

2.4.1 Physical Network Testbeds

Physical network testbeds employ real physical hosts and network hardware for conducting research experiments. Emulab (White et al., 2002) was a distributed physical network setup, implemented for conducting research experiments. It consists of 218 physical nodes distributed between two US universities. Netbed (White et al., 2002) is a simulation environment implemented on Emulab that provides time and space sharing and employs ns-2 for research and development. Emulab evolved into DETER (Benzel et al., 2007), which is a cluster based testbed, consisting of high end workstations and a control software. It uses high-performance VLAN-capable switches to dynamically create nearly arbitrary topologies among the nodes. It was the first testbed to be remotely accessible through the public Internet infrastructure. The 1998 DARPA off-line intrusion detection evaluation (Lippmann et al., 2000) and LARIAT (Rossey et al., 2002) are also two physical network testbeds sponsored by US Air Force and developed at the Lincoln Laboratory, MIT.

2.4.2 Simulation Testbeds

Simulation testbeds employ simulation tools to conduct network experiments. PDNS and GTNetS (Perumalla and Sundaragopalan, 2004) were two network simulators for developing packet level worm models. These simulators allow an arbitrary subject network configuration to be specified consisting of scan rate, topology and background traffic. On the basis of defined input

parameters, various types of outputs such as number of infected hosts in any given instance, sub-millisecond granularity of network event statistics or a global snapshot of the entire system are produced. Ediger reported the development of the Network Worm Simulator (NWS) (Ediger, 2003), which implements a finite state machine concept to simulate network worm behaviour. Tidy et al (Tidy, Woodhead and Wetherall, 2013) have reported a large scale network worm simulator aimed at the investigation of fast scanning network worms and candidate countermeasures.

2.4.3 Emulation Testbeds

Emulation testbeds provide a compromise between simulation and real world testing. ModelNet (Vahdat et al., 2002) is a emulated testbed, implemented for general networking and distributed system experiments. In ModelNet, unmodified applications run on edge nodes, configured to route all their packets through a scalable core dedicated server cluster, by emulating the characteristics of a special target topology. Honeypots such as Honeyd (Provos, 2004) can also be classified as an emulation system as it has been used in many recent security systems for malware detection and capture.

2.4.4 Full System Virtualization Testbeds

Full system virtualization testbeds employ full virtualization; a technique that provides a type of virtual machine environment with complete simulation of the underlying hardware. vGround (Jiang et al., 2006) has extended UML's virtual networking capabilities by supporting a VM-create-VM approach to automatically extend the network size. It uses Snort (Snort, 1998) and Bro (Paxson, 1998) as NIDS and Kernort (Jiang, Xu and Eigenmann, 2004) as a HIDS to monitor worm target discovery and propagation. ViSe (Richmond, 2006) provides a virtualization platform where malware exploits can be tested against the entire range of x86 based operating systems under controlled conditions, while being monitored by a NIDS. V-NetLab (Sun et al., 2008) has implemented a model based on DETER's (Benzel et al., 2007) remote access capability by utilizing data link layer virtualization and packet encapsulation, thereby providing a more secure means of remote access to security related testbeds. Golath (Fagen, Cangussu and Dantu, 2009) is a virtual network based

on a Java Virtual Machine (JVM) and the Ultra-light-weight abstraction level (ULAL). It provides a virtual environment to run any application written in Java, independent of the type of host operating system. Host behaviour can be monitored in this environment by adding different Java plug-in extensions.

2.5 Mathematical Models for Worm Propagation

Mathematical models for worm propagation helps us understand the epidemiology of worm outbreaks (Chen and Robert, 2004), (Moore et al., 2003), (Staniford et al., 2004). Various authors have proposed mathematical models to describe worm propagation (Chen and Robert, 2004), (Moore et al., 2003), (Staniford et al., 2004), (Zou, Gong and Towsley, 2002), (Liljenstam et al., 2003); based on the models originally developed for biological epidemiological studies (Kermack and McKendrick, 1927), (Frauenthal, 1980). The susceptible-infected (SI) (Kermack and McKendrick, 1927) model is the most widely reported biological model, which models the epidemiology of infection by assuming a population of hosts is of fixed size and relying on a deterministic contact coefficient to govern the differential between each step of the model. Variations of the SI model in the field of biological epidemiology tend to add addition states (Frauenthal, 1980), for example the susceptible-infected-recovered (SIR) model in which all hosts stay in one of only three states at any time: 'susceptible' (denoted by 'S'), 'infectious' (denoted by 'I') or 'recovered' (denoted by 'R'). The susceptible-infected-susceptible (SIS) is another variation on the SI model that adds the ability of an infectious host to transition back the susceptible state. Of note is another work undertaken by Chen et al. (Chen, Gao and Kwiat, 2003) that reports a discrete time deterministic model of active worms (the AAWP model), which characterizes the propagation of worms that employ random scanning and local subnet scanning. It uses a discrete time model and a deterministic approximation to describe the spread of computer worms.

2.6 Research Questions

2.6.1 Research Question 1

All the proposed solutions set out in section 2.3 of this chapter provide potential or partial countermeasures against network worms. The following are limitations in the above mentioned classes of solutions:

- RL or containment solutions (as described in section 2.3.1) are limited in the efficient and effective detection of worms and lack the functionality to spread malware warnings to unaffected networks.
- ASG solutions (as described in section 2.3.2) are prone to a high no of false positives. They also lack the functionality to spread malware warnings to unaffected networks.
- BSG solutions (as described in section 2.3.3) lack the distributed worm detection and containment function that is effective in the case of fast spreading worms.
- LA solutions (as described in section 2.3.4) lack effective worm detection capabilities.
- PP solutions (as described in section 2.3.5) are prone to false positives, impractical to implement in a real network and lack the functionality to spread malware warnings to unaffected networks.
- MC solutions ((as described in section 2.3.6) are not efficient in the case of fast scanning, flash or hit-list worms due to their bandwidth usage, legality and limited zero- day vulnerability patching capabilities.
- HQD solutions (as described in section 2.3.7) lack an efficient mechanism for worm detection.
- Defensive worms (as described in section 2.3.8) are ineffective in the case of fast zero-day scanning flash and hit-list worms due to their limited zero day vulnerability patching capabilities and legality to release on the Internet.

From the above list, it is clear that none of these solutions, in isolation, provides an effective and efficient approach for zero day worm detection and

containment in a disturbed environment. Hence, research question 1 is defined as follows:

- ***Is it possible to develop and evaluate a distributed, automated worm detection, prevention and containment solution that will be more effective against fast zero-day worms than the potential solutions summarised in section 2.3? Such a countermeasure may be limited to adding delay to the worm infection time so that system administrators have additional time to patch infected hosts. It would be desirable for such a countermeasure to be able to stop the worm infection completely.***

2.6.2 Research Question 2

To the knowledge of the author, no previous research has reported the design and development of worm daemon which works in a similar way to a random scanning and a hit-list worm such as SQL slammer and Witty, which is self-contained within an isolated environment, which is self-configurable with speed of propagation and hit-list. Hence there is a need to design and develop a worm daemon, which can be employed to empirically investigate the spread of a random scanning and hit-list worm in an isolated environment with real world Slammer or Witty exploitable conditions and also to test potential countermeasures.

Hence in order to address above limitations and characterising the virulence of worms, the following research question is defined:

- ***Is it possible to develop a pseudo worm daemon with characteristics such as random and hit-list scanning, configurable rate of propagation and confinement within defined network space to allow a developed countermeasure to be empirically tested and evaluated?***

2.7 Chapter Summary

This chapter has presented the definitions of different types of malware, a taxonomy of computer network worms and details of potential wormable vulnerabilities. It has also summarised a wide range of previously reported worm detection and prevention mechanisms, worm testing environments and mathematical models for worm propagation, and classified them into different

categories. Finally, two research questions have been defined based on the limitations identified in the existing work. The next chapter will present the details of a proposed distributed worm detection and containment countermeasure.

3 THE RATE LIMITING + LEAP AHEAD (RL+LA) SCHEME

3.1 Chapter Introduction

The Rate Limiting + Leap Ahead (RL+LA) scheme is designed as a worm detection and containment scheme, which is then implemented as a proof-of-concept in the C programming language (Shahzad and Woodhead, 2014a). The source code of RL+LA is given in Appendices of this thesis. The scheme can be deployed on the routers of enterprise networks. It uses the absence of a Domain Name System (DNS) (Mockapetris, 1987) lookup, prior to an outgoing TCP SYN or UDP datagram to a new destination IP address as a behavioural signature to detect worm scanning activity. Upon detection of such behaviour, the scheme blocks further traffic from the originating host at the network gateway and sends an alert message using a variation of the Friends protocol (Nojiri, Rowe and Levitt, 2003) to peer routers which belong to the scheme. To the author's knowledge, this is the first implementation of a hybrid worm detection and containment mechanism based on a combination of Rate Limiting (RL) on the basis of behaviour signature detection and Leap Ahead (LA) solutions. The novelty of this scheme is: its automated, distributed behaviour based worm detection, containment and alerting to participating peer networks in the scheme. A hybrid worm detection and containment solution was designed and implemented as none of solutions described in section 2.3, in isolation, provides an effective and efficient approach for zero-day worm detection and containment in a disturbed environment.

3.1.1 Chapter Layout

This chapter starts by presenting the basis concept of the RL+LA scheme in section 3.1. Section 3.2 defines the basic design and methodology of RL+LA scheme. Section 3.3 discusses the RL+LA system design and implementation by presenting its algorithm while section 3.4 provides the concluding statement.

3.2 Basic Design and Methodology

The DNS (Mockapetris, 1987) is a hierarchical globally distributed database for computers, services or any resource connected to the Internet that translates easily memorized domain names to the numerical IP addresses needed for the purpose of locating computer services and devices worldwide. It can be classified as phone book for the Internet by translating human-friendly computer hostnames into IP addresses. Almost all network traffic leaving a workstation host for another Internet host, with which it has not recently communicated, requires a DNS lookup. It is quite usual for network segments to be logically or physically separated in an enterprise network due to various reasons including administration, security, geographical location etc. Whyte et al. (Whyte, Kranakis and Oorschot, 2005) divides the different network segments into cells as shown in Figure 3.1. According to Whyte et al., the traffic generated by the propagation of fast scanning worms can be considered under the following three classifications:

- ***Local to local (L2L):*** In L2L, scanning worm targets hosts within the boundaries of the enterprise network in which the source host resides. Topological worms employ this method to propagate.
- ***Local to remote (L2R):*** L2R refers to a scanning worm whose source host is within an enterprise network but which is targeting the whole Internet.
- ***Remote to local (R2L:*** While in R2L propagation, scanning worms target hosts within an enterprise network from elsewhere within the Internet.

The proposed worm detection and containment scheme: The RL+LA, detects the L2R propagation of worms based on behaviour signature (lack of DNS lookup), and alerts other peer networks, using a variation of the Friends protocol (Nojiri, Rowe and Levitt, 2003) of the detected worm event. Ganger et al. (Ganger, Economou and Bielski, 2002) first proposed that the lack of DNS lookup from a host might be used as a tell-tale sign of worm scanning activity. In the case of a worm infection like Slammer (Moore et al., 2003), an infected

host tries to send as many UDP datagrams as it can, per unit time, without making any DNS requests. The RL+LA scheme uses this behavioural signature (lack of DNS lookup) as an indicator of worm scanning activity and alerts other participating peer networks.

Figure 3.1 shows the placement of the elements of proposed RL+LA scheme in an enterprise network. The RL+LA prototype is deployed on the internal network gateways of each cell, on the DMZ gateway to implement rate limiting and to send internal Friends messages in case of worm scanning activity. While RL+LA on the border gateway of each enterprise does not implement rate limiting, it only forwards the Friends messages to external Friends peers on the Internet if a worm malware warning is received. Each host in any network cell is allowed to send up to N outbound TCP SYN or UDP datagrams without a corresponding DNS lookup in a unit interval of time. If a host sends more than a threshold value N, outbound datagrams without appropriate DNS lookups in a specified time interval, the RL+LA implementation flags this as a worm infection indicator, uses iptables (The netfilter.org "iptables" project, 1998) to block further datagrams from the host from exiting the network cell locally, reduces the threshold to N/2 and sends an alert message to internal and external peers using the Friends protocol. On receipt of such a message, each peer will reduce its trigger threshold to N/2.

3.3 The RL+LA: System Design and Implementation

The RL+LA proof-of-concept implementation is coded in the C programming language. The C programming language was chosen to implement RL+LA prototype due to its capability to access the system's low level functions and easily available open source libraries like libpcap (TCPDUMP & LIBPCAP, 2008) and libpjlib (PJSIP: PJLIB Library, 2008). The libpcap library is used to capture traffic and the libpjlib library is used for parsing incoming DNS replies. Figure 3.2 shows the flowchart of the RL+LA algorithm. For any TCP SYN or UDP datagram leaving the network, RL+LA looks for a corresponding DNS lookup in Table A: Network DNS Cache. In the absence of a corresponding entry, it adds the source IP address to Table B: Counters and increments the counter value. The result of all DNS lookups along with the

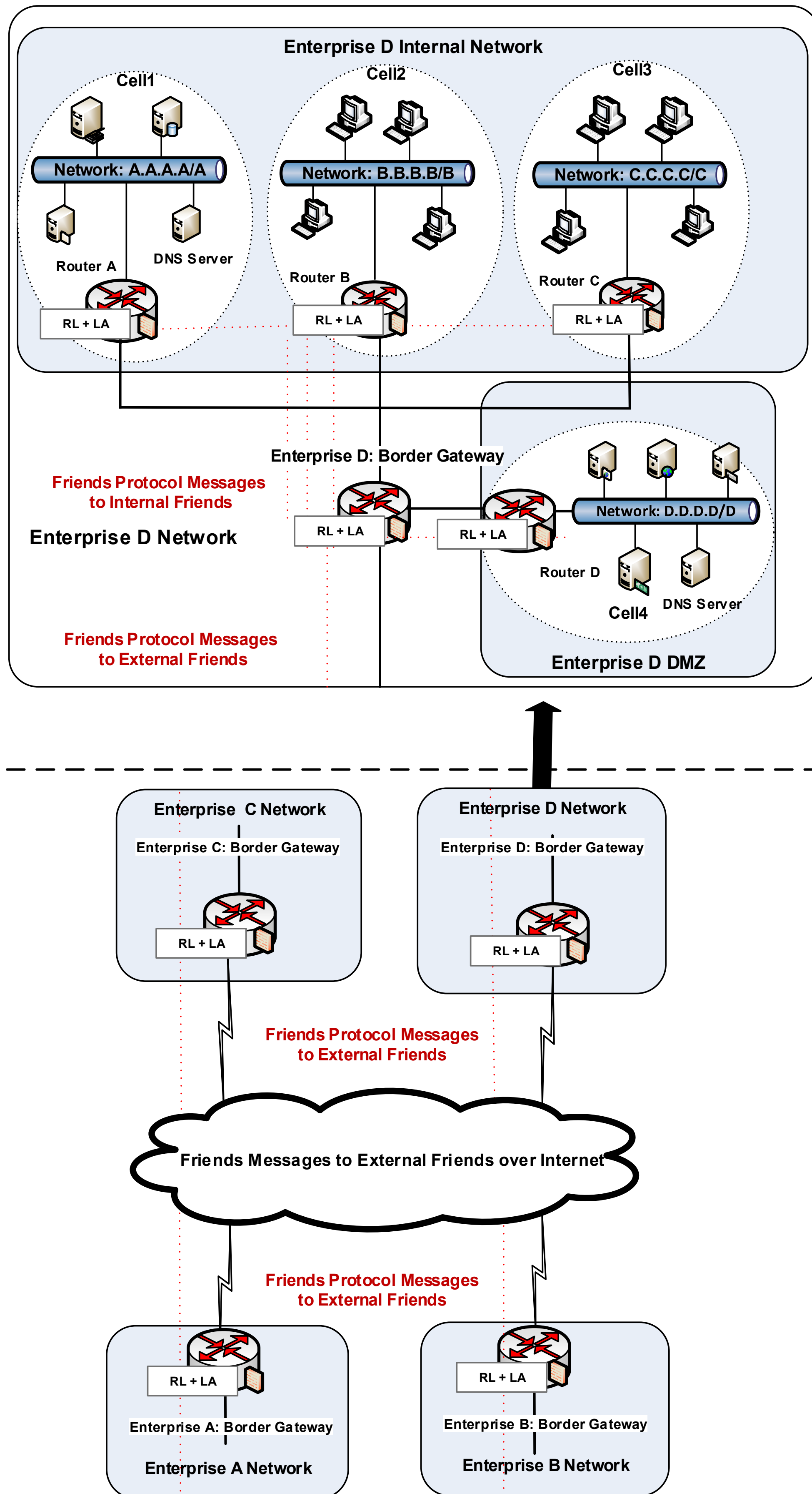

Figure 3-1 The RL+LA Proposed Design Architecture

source IP address and the destination IP address is saved in Table A. Different threshold values can be defined for different networks, depending on the nature of the typical traffic of that network. Another time interval K is defined in Table B to decrement the values in Table B. The higher the rate of decrementing the value of K in Table B, the lower the probability of a false positive being triggered.

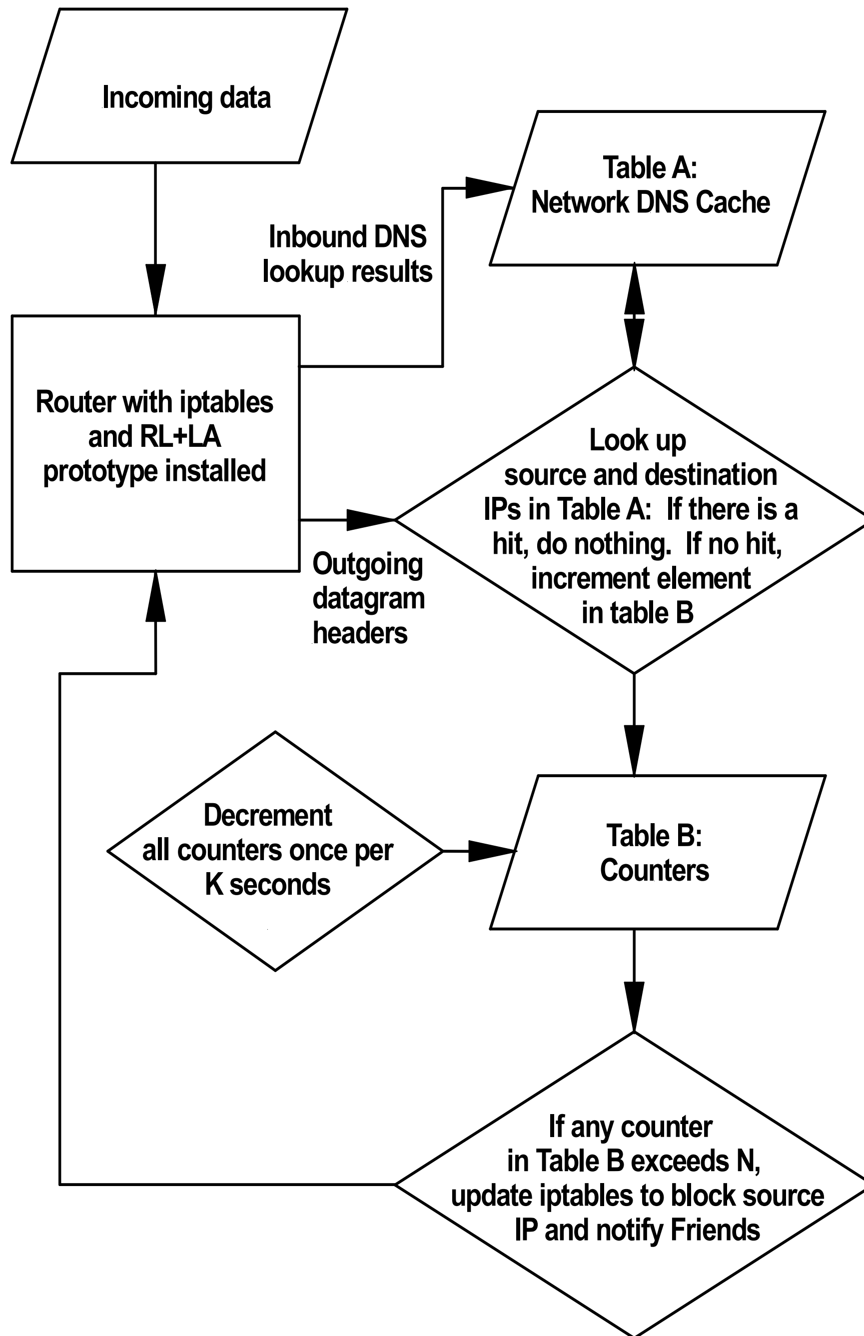

Figure 3-2 Flow Chart for The RL+LA Prototype Algorithm

It should also be noted that some legitimate network services generate UDP datagrams without a preceding DNS lookup, such as the DNS service itself. In order to address this situation, a small number of destination IP addresses are white listed in the system, such as those for the primary and secondary DNS servers, and other can be added.

Once the threshold value is reached in Table B, the RL+LA application blocks outgoing traffic from the offending host using iptables, reduces N to N/2 and sends an alert message using the Friends protocol to internal peer routers and to the border gateway, which in turn forwards the alert to external peers in the scheme, again using the Friends protocol. Each alert message contains the router user name, a predefined password, and a command to half the threshold value in Table B.

3.4 Chapter Summary

This chapter has presented the architecture and design of the RL+LA scheme. The RL+LA scheme uses the absence of a DNS lookup, prior to an outgoing TCP SYN or UDP datagram to a new destination IP address as a behavioural signature to detect worm scanning activity and then uses the Friends protocol to send alert messages to other friends within the participating domain. This scheme is subject to experimental verification in order to evaluate its suitability in the case of a worm outbreak on a small scale (to show proof of concept) and on a large scale networks. In order to achieve these goals, a pseudo-worm daemon and a suitable worm countermeasure testing environment are required, which will be presented in the following chapter.

4 THE PSEUDO-WORM DAEMON (PWD)

4.1 Chapter Introduction

The Pseudo-Worm Daemon (PWD) is designed to empirically evaluate the developed countermeasure as set out in research question 2 in section 2.6.2. The PWD performs random scanning and hit-list worm like functionality, and is implemented as a proof-of-concept in the C programming language (Shahzad and Woodhead, 2014b). The source code of PWD is given in Appendices of this thesis. The C programming language was chosen to implement PWD prototype due to its capability to access the system's low level functions and easily available open source libraries, which makes it platform independent. This PWD prototype can be deployed on any host in an enterprise network and it functions in similar way to any random scanning and hit-list worm. As reported in section 2.6.2 (to the knowledge of the author), no previous reported research has presented the architecture and design of any worm daemon which works in a similar way to a random scanning and a hit-list worm such as SQL slammer, Witty etc, which is self-contained within an isolated environment, which is self-configurable with speed of propagation and contains a user defined hit-list. Hence, the novelty of this worm demon is its UDP based propagation, user-configurable random scanning pool, ability to contain a user defined hit-list, authentication before infecting vulnerable host and efficient logging of time of infection.

4.1.1 Chapter Layout

This chapter begins by introducing the basic concept of PWD in section 4.1. Section 4.2 presents the basic design and methodology of a random scanning or a random scanning hit-list worm. Section 4.3 reports the system design and architecture of the PWD, while section 4.4 explores its key characteristics. Section 4.5 reports the evaluation of PWD by using Pseudo-Slammer and Pseudo-Witty worms and the SI model, while section 4.6 by way of background, presents the Virtualized Malware Testbed (VMT), which is designed to evaluate PWD and RL+LA. Finally section 4.7 presents the chapter summary.

4.2 Basic Design and Methodology of Pseudo-Worm Daemon (PWD)

A random scanning worm such as Slammer, Code Red etc. (Moore et al., 2003), (CERT:Code Red II, 2001) uses a pseudo random number generator to scan random IP address whereas a hit-list worm such as Witty (Shannon and Moore, 2004) uses an initially generated hit-list embedded into it to infect vulnerable hosts on the Internet. Upon initial infection, a UDP based random scanning worm such as Slammer generates a number of UDP datagrams, whereas a TCP based worm such as Code Red initiates a number of connections, (defined by an attacker in the worm algorithm) and sends them to number of random IP addresses in a unit interval of time. Each new infected host, upon infection, follows the same process and starts scanning further IP addresses, thereby creating a chain reaction. Figure 4.1 shows this worm infection process (Chen and Robert, 2004).

In each stage of infection, each infected host n, further scans m hosts. In the case of a random scanning worm such as Slammer, at the first stage of infection, one or two hosts starts the infection process, while in the case of a hit-list such as Witty, at the first stage of infection, the worm contains an initial list of vulnerable hosts. It is to be noted that an already infected host can receive multiple copies of either UDP datagrams or TCP packet scans, as shown by dotted arrow in figure 4.1.

4.3 Pseudo-Worm Daemon System Design and Implementation

The PWD is implemented in the C programming language. The C programming language was chosen to implement the PWD due to its capability to access the systems low level functions, easily available open source compiler and ease of use. The basic design of PWD consists of three key elements:

- **UDP Server:** The UDP server program is a single threaded application and performs pseudo worm like functionality. It can be installed on any platform host. Upon receiving a UDP datagram from a UDP client on a user-defined port number and IP address, it looks for authentication

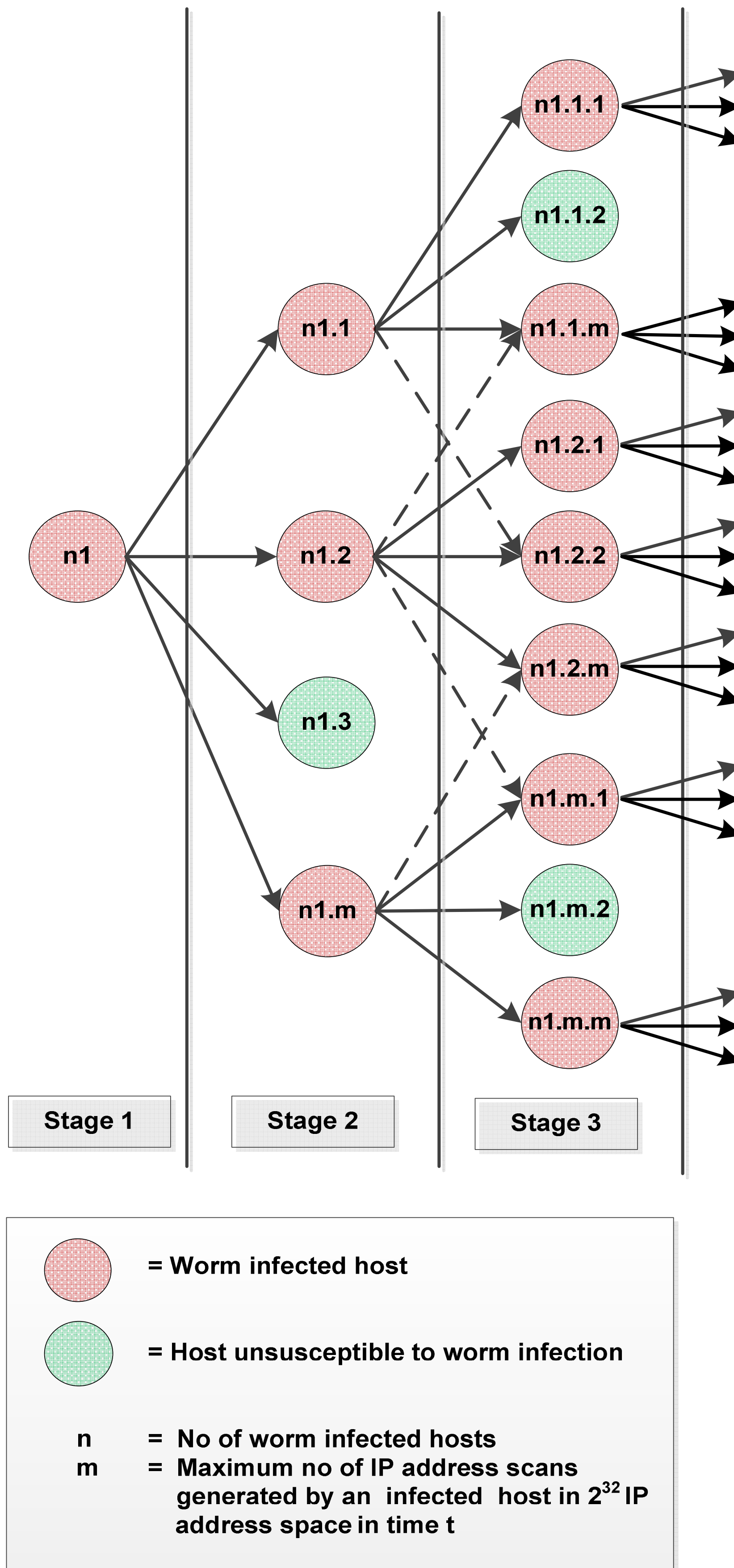

Figure 4-1 Worm Infection Process (Chen and Robert, 2004)

string (user- defined) to authenticates the UDP client request, and upon authentication, sends a UDP datagram containing local time of infection to logging server and turns its behaviour to that of a client by sending further UDP datagram to different destination IP addresses (generated by a using pseudo random number generator or read from text file already containing a list of vulnerable hosts). The rate of UDP datagrams generated per second and the pool from which random destination IP addresses are chosen (either by random scanning or hit-list from local file) are user-configurable parameters.

- **UDP Client:** A UDP client program is used to launch the worm. It can be installed on any platform host. It sends a UDP datagram to UDP server with IP address of UDP server, port number on which UDP server running and authentication string of UDP Server. UDP client in used only once to start the worm infection process.
- **Logging Server:** The logging server program is installed on any platform host in a network to log the time of infection from UDP servers on the network. All hosts running the UDP server holds the IP address of the central logging server and upon infection, sends the time of infection to the logging server.

The following figure 4.2 shows design architecture of PWD. It can be seen that there in only one UDP client presented, which is used to start the worm infection process. After that, the UDP Server performs the functionality of a true random scanning or hit-list worm.

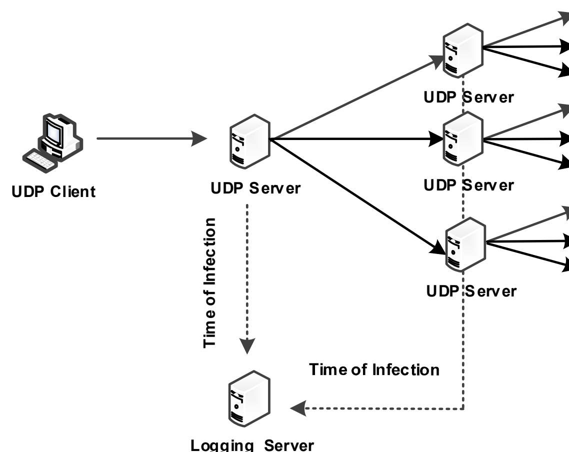

Figure 4-2 Design Architecture of PWD

The following figure 4.3 shows flow diagram of the PWD algorithm, that describe its process for only one instance of worm.

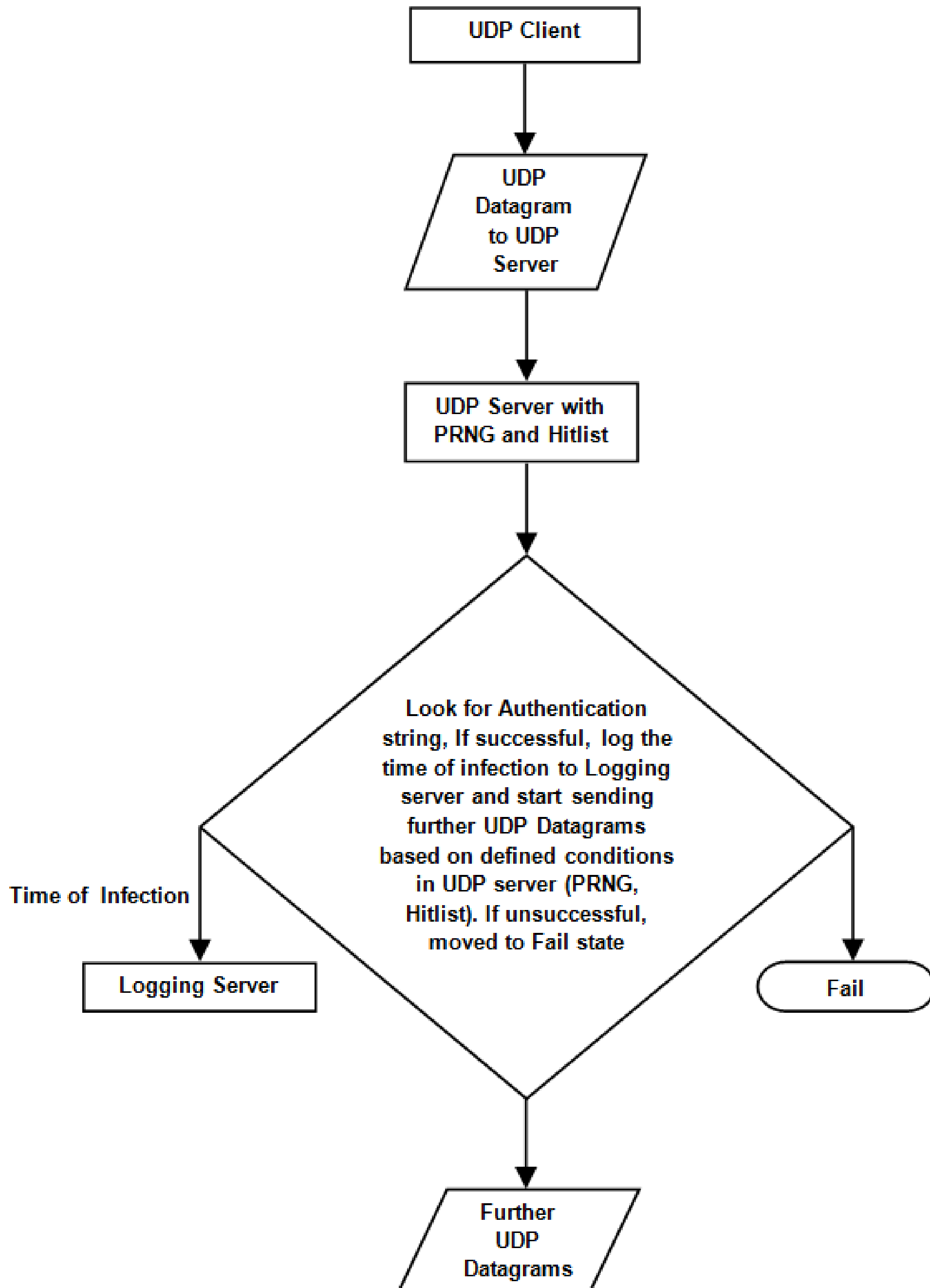

Figure 4-3 Flow Diagram of PWD Algorithm

4.4 Characteristics of the Pseudo-Worm Daemon (PWD)

Following are key characteristics of the Pseudo-Worm Daemon (PWD).

4.4.1 UDP based Propagation

The designed PWD uses UDP as its propagation mechanism, thereby making it similar in functionality to the SQL Slammer and Witty worms. A UDP based worm can propagate much faster than a TCP based worm (Staniford et al., 2004), due to the fact that TCP based worm uses three way handshake for connection establishment before infecting a new host, whereas a UDP based worm uses a single datagram to infect another host. SQL Slammer is considered to be the fastest random scanning worm in history as its infected population doubled in size every 8.5 seconds, with 90 % of vulnerable host infected within 10 minutes (Moore et al., 2003).

4.4.2 Pseudo Random Number Scanning

The PWD implementation prototype presented in this chapter uses pseudo random number scanning to generate new IP addresses. A pseudo random number generator (PRNG) is an algorithm for generating a sequence of numbers that approximates the properties of random numbers (Marsaglia, 2003). A random seed is used to initialize the PRNG. Various types of PRNG exist, but the PWD implementation prototype presented in this chapter, uses a Complementary-multiply-with-carry (CMWC) type of pseudo random number generator (Marsaglia and Zaman, 1991). CMWC method generates sequences of random integers based on an initial set from two to many thousands of randomly chosen seed values. The key advantages of the MWC method are: (a) it invokes simple computer integer arithmetic, (b) leads to very fast generation of sequences of random numbers with immense periods, ranging from around 2^{60} to $2^{2000000}$.

4.4.3 Hit-List

A pre-generated list of vulnerable IP address can be provided to the PWD in the form of a text file. The PWD reads the file and sends a single UDP datagram to those IP addresses on a specific port, thereby imitating the functionality of a hit-list worm such as the Witty.

4.4.4 Containment

The PWD has a user-configurable random scanning IP addresses pool, which can be defined inside it code. For examples, generating IP addresses in one class A or generating IP addresses in six class C networks or generating IP addresses over the whole Internet space. Hence, its random IP generation can be contained in any network size according to the needs of the experiment.

4.4.5 Scanning rate

The PWD can be configured to scan at different scanning rates. For example, 100 scans per seconds, 500 scans per seconds etc. The number of random IP addresses scanned per second is defined as the scanning rate of the worm. For example, on average, Slammer was reported to have scanned 4000 IP addresses per second.

4.4.6 Authentication

An authentication mechanism is included into the PWD for safety reasons. Any UDP datagram from the PWD contains an authentication string. Upon receiving a UDP datagram, a host looks for authentication string, and if it finds the authentication string, it starts scanning new hosts.

4.4.7 Logging and Reporting

The PWD prototype also includes a logging server, which can be installed on any host. Upon infection, the UDP server sends the IP address of the newly infected host and the time of infection (with resolution of 10^{-6} seconds) to the central logging server. The central logging server stores this information in a text file which can be processed to extract the time of infection of all vulnerable hosts on the network.

4.5 Evaluation of Pseudo-Worm Daemon (PWD)

In order to evaluate the effectiveness of PWD as an effective tool to empirically analyse the propagation behaviour of random scanning and hit-list worms, and to test potential countermeasures, a Virtualized Malware Testbed (VMT) has been setup (which is given as background in section 4.6) and a series of experiments were conducted using the real worm attributes of

Slammer and Witty worms. The SQL Slammer and Witty worms were selected for evaluating the PWD due to the fact that reliable empirical data from both worm events are available from CAIDA (CAIDA: Center for Applied Internet Data Analysis, 2014). Furthermore, Pseudo-Slammer and Pseudo-Witty worms results are compared using the SI model. Hence, the effectiveness of PWD was evaluated by using two ways:

- Comparing Pseudo-Slammer and Pseudo-Witty worms results with real outbreak data (which is available from CAIDA).
- Mathematically modelling the Pseudo-Slammer and Pseudo-Witty worms results by using SI model and comparing the infection process.

4.5.1 Pseudo-Slammer Worm Experiments

4.5.1.1 Slammer Worm Outbreak Attributes

Moore et al (Moore et al., 2003) reported some key characteristics of the Slammer outbreak of 2003 which can be summarised as follows:

- 18 hosts per million of the entire IPv4 address space were susceptible to infection.
- The maximum recorded scanning rate of Slammer was 26,000 datagrams per infected host per second. This figure seems reasonable while considering the upper bound of 100BaseT interface and the worm Ethernet frame size of 430 bytes.
- The average scanning rate of Slammer was 4000 datagrams per worm instance per second during its entire infection period.

4.5.1.2 Experimental Setup

In order to empirically analyse the behaviour of the Slammer worm and to validate the PWD prototype on a class A scale, an experimental test network was configured on the Virtualized Malware Testbed (VMT) (reported in section 4.6 of this chapter), comprising of a single Class A address space 10.0.0.0/8 but divided into four subnets; 10.0.0.0/10, 10.64.0.0/10, 10.128.0.0/10 and 10.192.0.0/10 as shown in figure 4.1. These four subnets were connected through a central router by using RIP, configured on Quagga. Eight further Quagga based routers were implemented (two for each subnet). The RL+LA prototype was installed on each of these eight routers. The RL+LA prototype installed on routers A, B, C and D was configured to rate

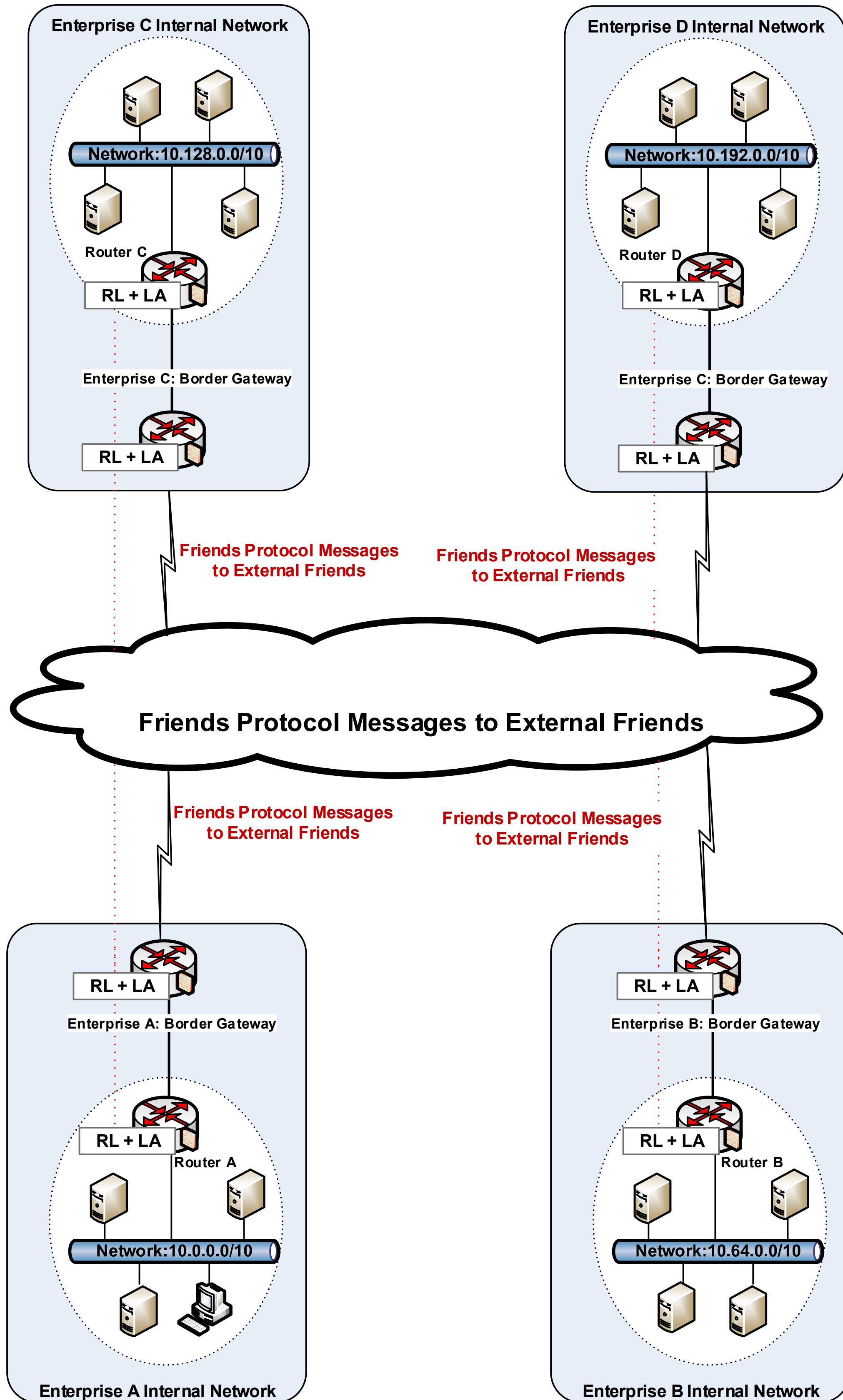

Figure 4-4 Slammer Worm Experimental Test Network

limit the outbound connection based on DNS anomalies and to send Friends protocol messages whereas the RL+LA prototype installed on the border routers only forwarded the Friends protocol messages received from internal and external friends. One Linux based virtual host was running in each subnet to provide a DHCP service and logging service for the Pseudo Worm Daemon (PWD). DSL was installed with the PWD on each of the susceptible virtualised hosts. All hosts in the network are time synchronized by using the Network Time Protocol (NTP).

4.5.1.3 Experimental Methodology

As reported in section 4.5.1.1, approximately 18 hosts per million of the entire IPv4 addresses space were susceptible to infection with Slammer and it achieved an average scan rate of 4,000 datagrams per infected host per second.

A single class A network has 2^{24} hosts, and so will contain $2^{24} * 18/1,000,000 = 302$ susceptible hosts. On this basis, 302 virtual hosts with the Slammer like pseudo-worm daemon were deployed across the four subnets. Each worm daemon was configured to scan within a single class A network (10.0.0.0/8). In order to avoid overloading the server farm hardware (in which case the experiments would have been measuring the effect of the hardware restrictions, rather than the properties of the worm), the average worm scanning rate was scaled down by a factor of 80. Therefore, based on an average scan rate reported by Moore et al. of 4000 scans per second, the Pseudo-Slammer network daemon was configured to scan at 50 scans per second in the set of Slammer experiments.

4.5.1.4 Experimental Results

Figure 4.6 shows the results of a set of three experiments conducted without implementing any countermeasures. In the first experiment, all 302 susceptible hosts were infected in 15.07 minutes. In the seconds experiment, all 302 susceptible hosts were infected in 14.58 minutes. While in the third experiment, all 302 susceptible hosts were infected in 14.45 minutes.

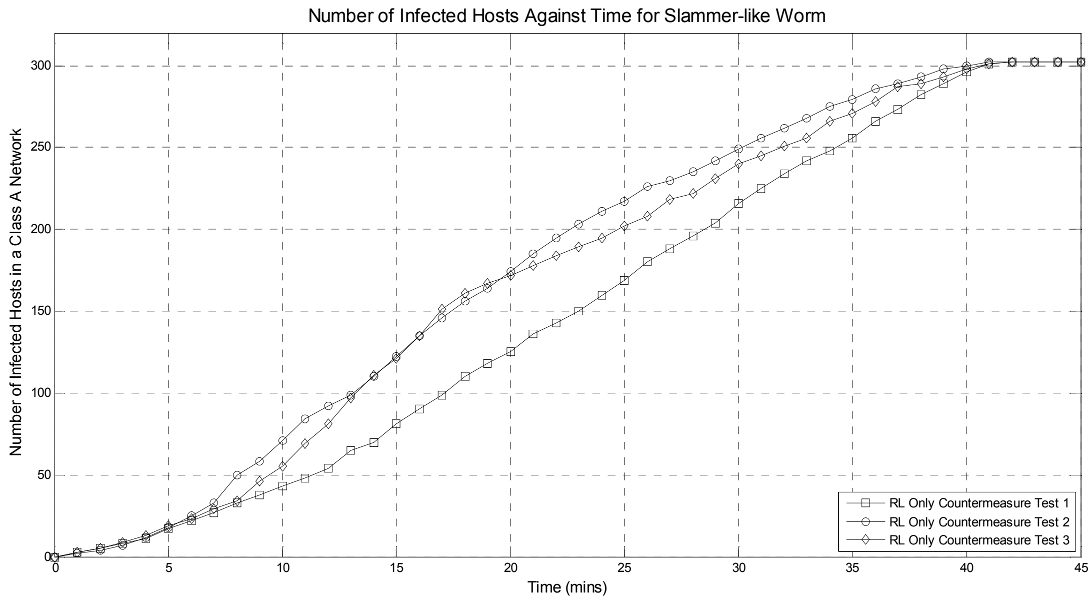

Figure 4-5 Experimental Results of Pesudo-Slammer Worm

4.5.2 Pseudo-Witty Worm Experiments

4.5.2.1 Witty Worm Outbreak Attributes

Shannon et al. (Shannon and Moore, 2004) reported some key characteristics of the Witty worm outbreak of 2004 which can be summarised as follows:

- The Susceptible population of the Witty worm was 12, 000or between 2 and 3hosts per million of the entire IPV4 address space.
- Witty worm had a variable datagram size, with an Ethernet frame size between 796 and 1307 bytes.
- The average scanning rate of Witty was 357 datagrams per infected host per second during its entire infection period while the maximum recorded scanning rate was 970 datagrams per host per second.
- Witty also utilized an initial hit-list of 110 hosts which were reported to have been infected in the first 10 seconds of launch. Of these110 hosts, 38 hosts were transferring 9700 datagrams per host per second continuously for a period of an hour.

4.5.2.2 Experimental Setup

In order to empirically analyse the behaviour of the Witty worm and to validate the PWD prototype, an experimental test network was configured on the Virtualized Malware Testbed (reported in section 4.6 of this chapter),

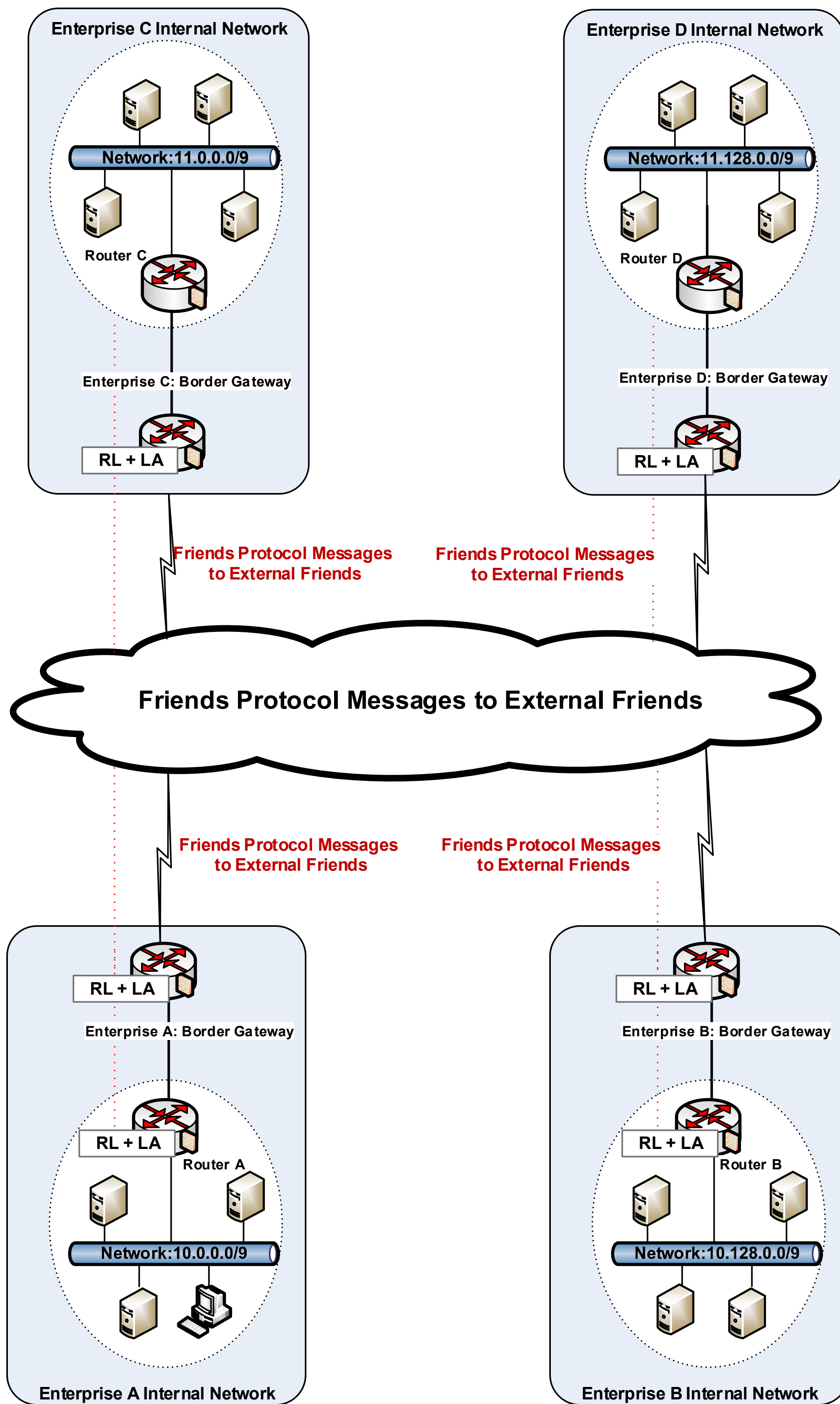

Figure 4-6 Witty Worm Experimental Test Network

comprising of a two Class A address space 10.0.0.0/8 and 11.0.0.0/8 but divided into four subnets; 10.0.0.0/10, 10.128.0.0/9, 11.0.0.0/9 and 10.128.0.0/9 as shown in figure 4.7. All the other network elements of experimental test network were the same as those defined previously in section 4.5.1.3.

4.5.2.3 Experimental Methodology

As reported in section 7.3.1, Witty had 3 hosts per million of the entire IPv4 addresses space were susceptible to infection with an average scan rate of 357 datagrams per infected host per second.

A single class A network has 2^{24} hosts, and so 2 class A networks will contain $2^{24} * 2(3/1,000,000) = 101$susceptible hosts. On this basis, 101virtual hosts with the Witty like pseudo-worm daemon were deployed across the four subnets. Each worm daemon was configured to scan within two class A networks (10.0.0.0/8, 11.0.0.0/8) at a scanning rate of 357 scans per host per second while using an initial hit-list of one susceptible host held by the first infected host.

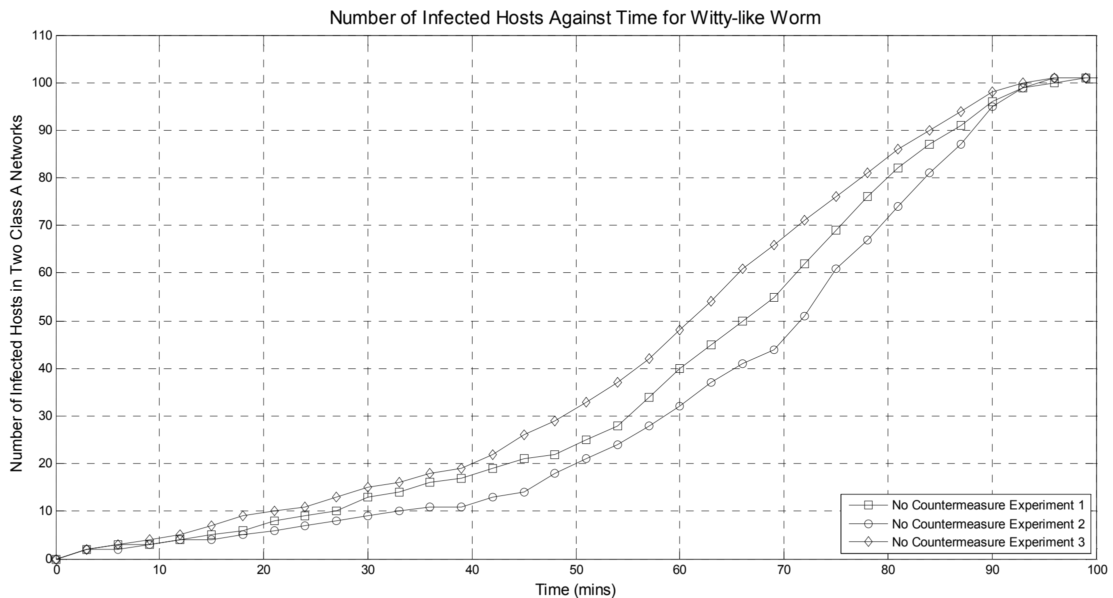

Figure 4-7 Experimental Results of Pesudo-Witty Worm

4.5.3 Discussion

4.5.3.1 Empirical Analysis of Pseudo-Slammer Worm Results

Figure 4.9 shows a comparison of real Slammer worm outbreak of 2003 with the results of the Pseudo-Slammer worm experiments. The average data for three Pseudo-Slammer worm experiments is plotted against the real outbreak of 2003 where empirical data is only available for the first 4 minutes of infection (Moore et al., 2003). The analysis conducted by Moore et al. states that the real slammer worm infected more than 90 percent of vulnerable hosts within 10 minutes (Moore et al., 2003). It is also observed from the Pseudo-Slammer experiments conducted on the VMT platform that all three experiments achieved infection of 90% of vulnerable hosts within approximately 10 minutes, whereas 99% of infection is achieved in 14 minutes. Hence these experimental results are broadly comparable to the available data for the real Slammer outbreak of 2003.

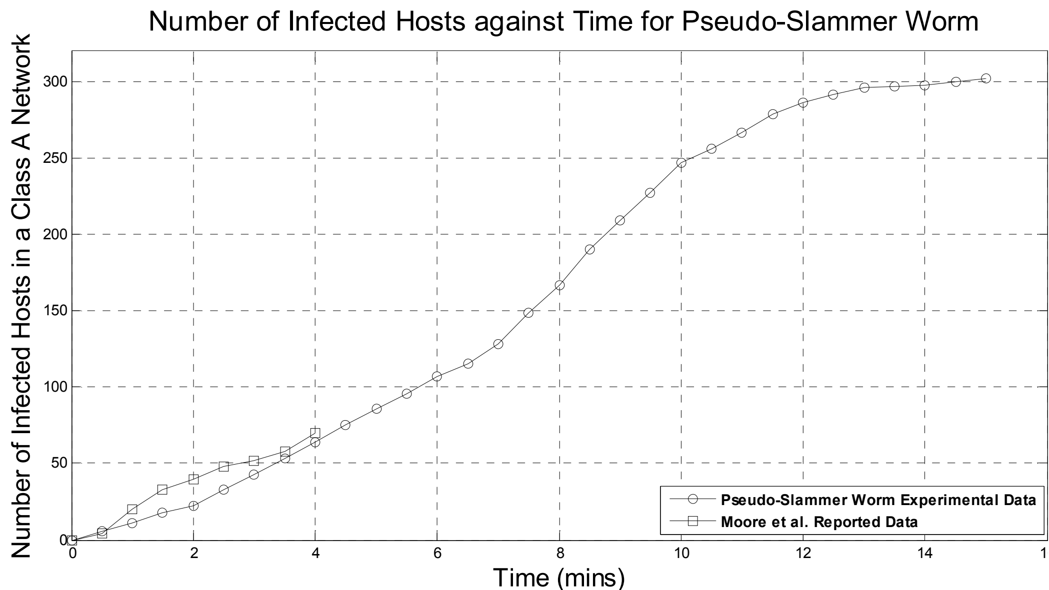

Figure 4-8 Pseudo-Slammer Experiments vs. Real Slammer Outbreak

4.5.3.2 Empirical Analysis of Pseudo-Witty Worm Results

Figure 4.10 shows a comparison of data from the real witty worm outbreak of 2004 with results of the Pseudo-Witty worm experiments. The average of the three Pseudo-Witty worm experiments is plotted against the real witty worm outbreak of 2004 as reported by Shannon et al. (Shannon and Moore, 2004).

Shannon et al. reported that the real Witty worm infected 90% of its susceptible hosts with within 90 minutes while 100 % of infection took almost 140 minutes. But, the Pseudo-Witty experiments conducted by using VMT took 90 minutes to reach its 90 % of infection and 97 minutes on average to infect all hosts. Furthermore, the infection process for real Witty Worm was quite fast at initial stage of worm spread. This difference is attributed to the fact that the real Witty Worm outbreak contained an initial hit-list of 110 hosts, out of which 38 infected hosts were transferring 9700 datagrams per host per second continuously for a period of an hour; whereas the Pseudo-Witty worm experiments used an average scan rate of the real Witty worm of 357 datagrams per host per second during its entire infection. The results of Pseudo-Witty worm experiments are still broadly comparable to the available data for the real Witty worm outbreak.

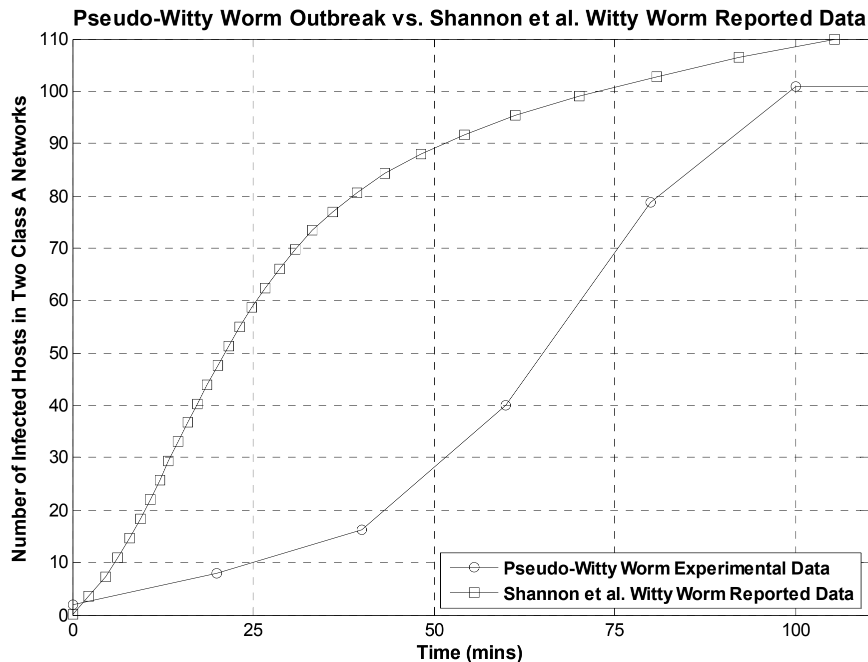

Figure 4-9 Pseudo-Witty Experiments vs. Reported Witty Outbreak

4.5.4 Epidemiological Modelling

4.5.4.1 Classical Simple Epidemic Model

In order to further analyse the spread of worm outbreaks, the reported research employed classical simple epidemic model (Kermack and McKendrick, 1927), (Daley and Gani, 1999), (Xiang, Fan and Zhu, 2009), in which all hosts exist in one of only two states at any given time: 'susceptible' (denoted by '*S*') or 'infectious' (denoted by '*I*'), and thus it is also called the SI model. This model assumes that once a host is infected by a worm, it will stay in an 'infectious' state forever. For a finite population of size N, it could be defined by the following single differential equation 4.1.

$$\frac{dI(t)}{dt} = \beta I(t)[N - I(t)] \qquad \textbf{(4-1)}$$

Where I(t) denotes the number of infectious hosts at time t; and β = η (Average worm scan rate) / Ω (The size of a worm's scanning space) stands for the pair wise rate of infection in epidemiology studies (Daley and Gani 1999). At the beginning of the infection *(t=0), I(0)* hosts are infectious and the other *N − I(0)* hosts are all susceptible.

Let *i(t)* stands for the fraction of the population that are infectious at time *t*, and thus *i(t) = I(t)/N*, which yields *I(t) = N*i(t)*. Substituting *I(t)* in equation (4.1) with *N*i(t)* and then rearranging it leads to equation 4.2:

$$\frac{di(t)}{dt} = N\beta i(t)[1 - i(t)] \qquad \textbf{(4-2)}$$

Equation (8.2) has following general analytical solution:

$$i(t) = \frac{e^{N\beta(t-T)}}{1 + e^{N\beta(t-T)}} \qquad \textbf{(4-3)}$$

Which is the logistic equation. For early *t, i(t)* grows exponentially. For large *t*, *i(t)* converges from 0 to 1 (all susceptible hosts are infected). When t = 0,

$i(t) = i(0) = \frac{e^{-N\beta T}}{1 + e^{-N\beta T}} = \frac{I(0)}{N}$ yields $e^{-N\beta T} = \frac{I(0)}{N - I(0)}$. Therefore, a particular analytical solution of equation 8.2, given its initial conditions $i(0) = \frac{I(0)}{N}$ is as follows:

$$\boldsymbol{i(t)} = \frac{\boldsymbol{I_0}}{\boldsymbol{I_0} + [\boldsymbol{N} - \boldsymbol{I_0}]\boldsymbol{e^{-N\beta t}}} \qquad \textbf{(4-4)}$$

4.5.4.2 Modeling Methodology and Results

Best fit SI model curves were plotted against experimental test results of Pseudo-Slammer and Pseudo-Witty Worm and values of Pearson's correlation coefficient *r*, (Pearson, 1895) as well as the value of β for SI model were calculated. Different values of β were the tried to obtain the highest value of *r*. This basic technique is similar to that employed by Tidy (Tidy 2014).

The figure 4.11 shows the best fit SI model against Pseudo-Slammer Worm results, showing the values of β and Pearson's correlation coefficient *r*.

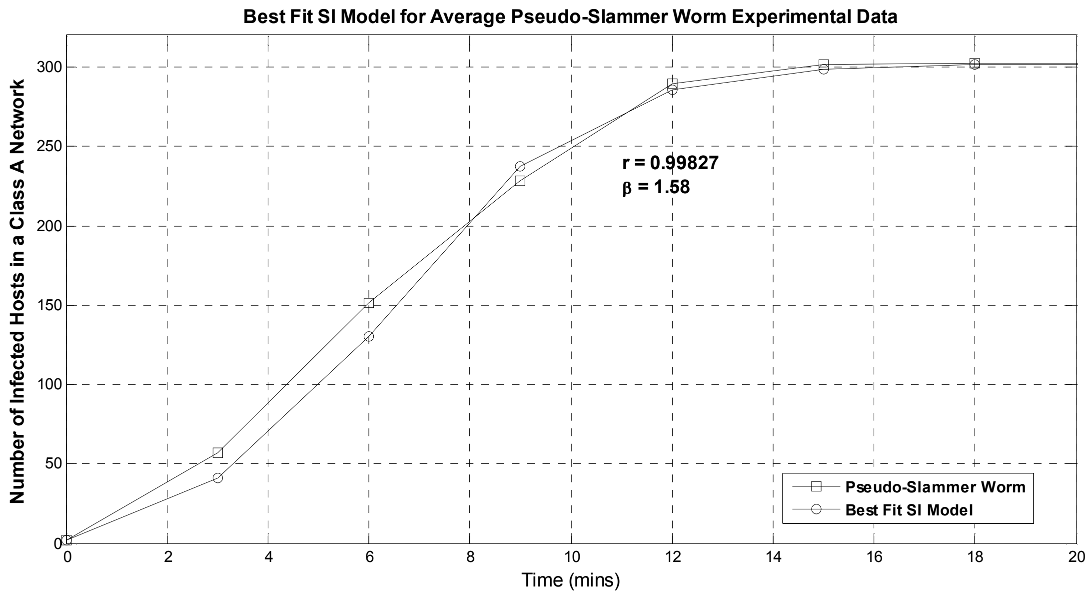

Figure 4-10 Best Fit SI Model for Pseudo-Slammer Worm Experimental Data

The figure 4.12 shows the best fit SI model against Pseudo-Witty worm results, showing the values of β and Pearson's correlation coefficient *r*.

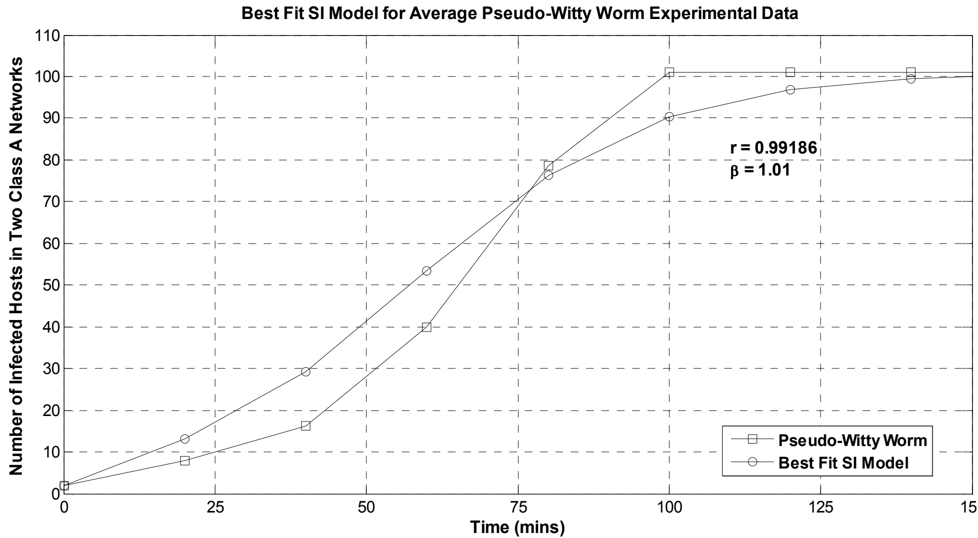

Figure 4-11 Best Fit SI Model for Pseudo-Witty Worm Experimental Data

- It is worthy of note that the value of the correlation coefficient, r, is quite close to 1 i.e. (r = 0.99186 as in case of Pseudo-Slammer and r = 0.99186 as in case of Pseudo-Witty), indicating the ability of the SI model to represent the experimental data for random scanning worms such as Slammer and Witty.
- Furthermore, obtained experimental results also proved that Pseudo-Slammer and Pseudo-Witty Worm outbreak follows random constant spread pattern and approximates to standard s-shaped curve as shown by Moore et al. (Moore et al., 2003).

4.6 Virtualized Malware Testbed (VMT)

This section presents the architecture, design and implementation of Virtualized Malware Testbed (VMT), which is included as background to chapter 4.

4.6.1 Introduction

Design and development of malware test environments for security experiments has been a key area of research over the last 10 years. Section 2.4 presented some existing malware testing environments. Based on the work, a

Virtualized Malware Testbed (VMT) (Shahzad, Woodhead and Bakalis, 2013) was setup by using virtualization technologies provided by VMware (VMware, 1998) and open source software such as Quagga (Quagga Routing Suite, 1999), Ubuntu (ubuntu, 2004), Damn Small Linux (Damn Small Linux (DSL), 2008). The key usage of VMT would be to conduct empirical experiments by using the PWD with real worm characteristics such as Slammer, Witty and contemporary potential worms such as those which might exploits Shell Shock (CVE:CVE-2014-6271, 2014) etc. in order to closely observe their infection and propagation behaviour. The same facility can also be employed in testing candidate worm countermeasures such as RL+LA (reported in chapter 3 of this thesis).

4.6.2 VMT Architecture Design and Implementation

VMT uses VMware ESXi (VMware ESXi, 2010) as the core virtualization technology. VMware ESXi is bare-metal embedded hypervisor that run directly on host server hardware without any additional underlying operating system. Various virtualization technologies such as Virtual Box, KVM, Xen etc. (Software Insider, 2013) exits but VMware was chosen as virtualization platform due to the following characteristics: ease of use, reliability, scalability of running virtualized hosts, remote administration of multiple servers from a single desktop host and vSphere PowerCLI for scripting administrative tasks. Damn Small Linux (DSL) (Damn Small Linux (DSL), 2008) was chosen to run as the virtualized operating system with the PWD. Although various other small Linux distributions such LINUXBBQ (LINIXBBQ, 2012), Puppy Linux (Puppy Linux, 2003), Tiny Core Linux (Tiny Core Linux, 2009) etc. but the main reason of selecting DSL as the virtual host operating system was its minimum hardware requirements. Each DSL based VM was configured with 32 MB of RAM and 1 GB of hard disk space, thereby making it a scalable solution with minimum reconfiguration time and ease of deployment.

VMT also uses a free and open source routing suite Quagga (Quagga Routing Suite, 1999) to provide a software routing functionality. This routing suite provides implementation of OSPFv2, OSPFv3, RIP v1 and v2, RIPng and BGP-4 for Unix platforms, particularly FreeBSD, Linux, Solaris and NetBSD.

Various open source software routing packages exists such as BIRD (The BIRD Internet Routing Daemon, 2008), GNU Zebra (GNU Zebra, 2005) but Quaaga was chosen as routing package due to its software support and ease of routing protocols configuration. It was installed on Ubuntu operating system (chosen due to ease of use)to provide routing functionality between different networks. Each Quagga based routing server, installed on top of Ubuntu used 2 GB of Ram and 5 GB of storage space. VMware vCenter Server (VMware vCenter Server, 2012) provides a graphical user interface to manage the VMware ESXi servers remotely. It also provides other functionality such as the ability to clone virtual hosts, virtual network configuration etc. It was installed on top of Windows Server 2003 R2. Ububtu based virtual hosts are also configured on which network services, such as DHCP, NTP and Logging server of PWD (reported in chapter 4.3) are configured. Each such virtual host image used 512 MB of RAM and 5 GB of disk space.

Figure 4.12 illustrates the physical architecture of VMT. It consists of a server farm with five servers, a management server, routing server with multiple network interface cards, Ethernet switches and external storage. Each server in the server farm is running ESXi while the management server is running VMWare vCenter Server, installed on top of Windows Server 2003 R2. One network interface card in each server farm host is connected to a logically isolated management network along with the management server; thereby allowing access to all resources from one graphical user interface. Multiple virtual topologies can be created within the server farm by using virtual local area networks (VLANs) connected to different NICs on the routing server, installed with Quagga. 1 TB external storage is also connected with the management server to take regular backup of the systems.

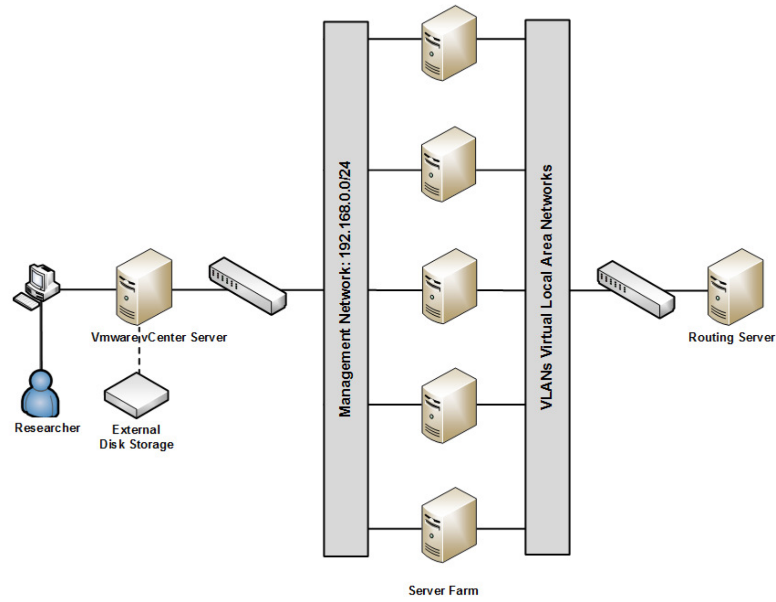

Figure 4-12 VMT Physical Network Setup

Table 4.1 summarizes the hardware and operating systems which make up the VMT infrastructure.

Table 4-1 VMT Hardware and Operating System Infrastructure

	Processors	No of cores	Operating System	Memory	Storage	VMs
Server 1	i7	6	ESXi 5.1	64 GB	1 TB	DSL, Ubuntu
Server 2	i7	4	ESXi 4.1	24 GB	1 TB	DSL, Ubuntu
Server 3	i7	4	ESXi 4.1	24 GB	1 TB	DSL, Ubuntu
Server 4	Xeon	4	ESXi 4.1	8 GB	512GB	DSL, Ubuntu
Server 5	Xeon	4	ESXi 5.1	8 GB	512GB	DSL, Ubuntu
Management Server	i7	4	Windows Server 2003 R2	8 GB	2 TB	N.A
Routing Server	i5	2	Ubuntu Quagga	4 GB	512GB	N.A

4.6.3 Characteristics of the Virtualized Malware Testbed (VMT)

Following are key characteristics of the Virtualized Malware Testbed (VMT).

4.6.3.1 Scale

The Virtualized Malware Testbed (VMT) reported in this chapter uses Damn Small Linux as the operating system for the PWD hosts, it is capable of running roughly 2000 virtual hosts, which can be deployed in 10 different subnets. But, in different scenarios as reported in chapters 4, 5 and 6 a maximum of 384 PWD based virtual hosts are configured according to the needs of the security experiments.

4.6.3.2 Cost

The VMT reported in this chapter can provide 2000 virtual hosts running PWD, with 10 fully routable networks. It has 3 i7 servers, 2 Xeon servers, 1 i7 management Server and 1 i5 routing host. The hardware costs of all these hosts do not exceed £5,000. The VMT used open source software and VMT products (VMware ESXi, VMware VCenter servers which are provided as part of VMware Academic Program (VMware Academic Program (VMAP), 2010) at nominal annual subscription fee of $250 to academic institutions.

In terms of the feasibility of scaling this architecture, a single i7 server can run 500 DSL based virtual hosts and can accommodate one or two class A networks. Hence 126 i7 servers can be used to create a network with all class A network address space 1.X.X.X-126.X.X.X, but with only 500 virtual hosts on each i7serve. Hence, it would be not be feasible to create an address space of the whole IPv4 Internet due to two reasons: (a) limitation of budget (b) using a larger network would not provide results with any greater value. A class A network has a 2^{24} host address space, which is sufficient enough to evaluate worm infection and to test potential countermeasure by using the experimental methodology described in sections 4.5.1.3 and 4.5.2.3. Hence, the experiments reported in sections 4.5.1 and 4.5.2 used one class A and two class A network address space to empirically analyse the Pseudo-Slammer and Pseudo-Witty worm respectively.

4.6.3.3 Flexible and Efficient Worm Experiment Control

A minimum rebuild and configuration time are key goals of any security testing environment. VMware vCenter Server provides PowerCLI (VMware vSphere PowerCLI 5.0, 2011); a command line interface tool that allows administrators to create simple and robust scripts to automate the main tasks, such as virtual hosts cloning, virtual hosts shutdown and reboot etc. PowerCLI shell scripts have been written to clone multiple virtual hosts. TheVMware vCenter Server graphical user interface also provides all of the above mentioned facilities.

4.6.3.4 Isolation

One network interface card in each server farm host is connected to a logically isolated management network (192.168.0/24) along with the management server, whereas multiple network interface cards are connected to routing servers with a different switch, thereby, completely isolating the VMT test network from management network.

4.6.3.5 Remote Administration

As VMT infrastructure uses the ESXi operating systems for all servers in the server farm of VMT, and uses VMWare VCenterServer installed on Management Server in order to access resources on all ESXi based servers. This provides remote administration of all servers in the server farm from single desktop host.

4.6.3.6 Confinement

As VMT uses PWD, which can be contained in defined networks according to the needs of the security experiment, and the Internet is completely isolated from the test network, the VMT provides the complete confinement of worm traffic within test networks. Furthermore, the PWD contains an authentication string to infect a host, which makes PWD traffic completely benign if it leaked on the Internet.

4.7 Chapter Summary

This chapter has presented the system architecture and design of the PWD. It has also reported its key characteristics such as UDP based propagation, pseudo random number scanning, ability to contain a user defined hit-list, user- configurable random scanning pool, configurable scanning rate,

authentication before infecting vulnerable hosts and efficient logging of the time of infection. Furthermore this chapter presents evaluation of PWD by (a) conducting a series of Pseudo-Slammer and Pseudo-Witty worm experiments with real outbreak attributes of Slammer and Witty worms; and comparing Pseudo-Slammer and Pseudo-Witty worms results with real outbreak data (which is available from CAIDA), (b) by mathematically modelling the results of the Pseudo-Slammer and Pseudo-Witty worms using the SI model and comparing the infection process. Finally, this chapter has presented (by way of background) the architecture and design of a Virtualized Malware Testbed (VMT), developed for worms testing, and based on VMware ESXi and open source softwares. It has also reported the key characteristics of VMT, such as scale, cost, flexible and efficient worm experiment control, isolation, remote administration, and confinement.

From this chapter, it is concluded that PWD can be used as an effective tool to empirically analyse the propagation behaviour of random scanning and hit-list worms, and to test potential countermeasures such as RL+LA (presented in chapter 3 of this thesis). However, in order to evaluate RL+LA, a comprehensive set of initial empirical experiments need to designed and performed, which will be presented in the following chapter.

5 EXPERIMENTAL RESULTS FOR THE RL+LA SCHEME ON A SMALL SCALE NETWORK

5.1 Introduction

This chapter builds on the work reported in the previous two chapters. Chapter 3 has presented the basic design and methodology of worm detection and containment scheme, The Rate Limiting + Leap Ahead (RL+LA), whereas chapter 4 has detailed the design and implementation of the Pseudo-Worm Daemon (PWD), it evaluation by conducting Pseudo-Slammer and Pseudo-Witty Worms and comparing the results with real worm outbreak data and SI model and the design and architecture the Virtualized Malware Testbed (VMT), designed to conduct security experiments in an isolated environment. The next step was to design and conduct a series of experiments in order to analyse the propagation behaviour of the PWD, and to analyse the performance of the proposed RL+LA countermeasure scheme, in comparison to other previously proposed countermeasure, such as RL only. Hence, a series of initial experiments were conducted using the PWD in VMT, to initially assess the effectiveness of the proposed RL+LA countermeasure. This chapter reports the experimental results of this series of initial experiments and a discussion of these results.

5.1.1 Chapter Layout

This chapter begins by presenting the experimental setup build for conducting the initial set of experiments in section 5.2. Section 5.3 details the experimental methodology used to conduct the experiments. Section 5.4 reports the results of the set of experiments by employing the defined methodology and experimental setup. Section 5.5 presents a discussion on the set of results with the need of future work. Finally section 5.6 concludes the chapter with a summary.

5.2 Experimental setup

To validate the RL+LA prototype, an experimental test network was configured in the VMT (reported in Chapter 4 of this thesis), consisting of six

fully routable class C networks (192.168.0.0 to 192.168.5.0) as shown in the figure 5.1.These six subnets were connected through three border routers (Router 1, 2 & 3) running Routing Information Protocol (RIP), configured on Quagga. Six further Quagga based routers (Router A-F) were implemented (one for each subnet). The gateway for each network ran a Linux 2.6 kernel along with iptables, the Quagga routing package and the RL+LA software. One Linux based virtual host was running in each subnet to provide a DHCP service, NTP service and Logging server of the PWD. Damn Small Linux (DSL) was installed with the PWD on each of the susceptible virtualised hosts. All hosts in the network were time synchronized by using the Network Time Protocol (NTP). Border router, Router 1 contained a list of external scheme peers (in this case Router 2 and Router 3). Internal routers (Routers A, B, C and D) exchanged Friends protocol alert messages directly in the case of worm scanning activity, whereas border router (Router 1) forwarded the alert messages to external scheme peers (Router 2 and Router 3). A network size of six class C networks was selected for experimentation due to undertake experiments on a small scale (scale was 6 class C networks) to begin with in order to get some initial sets of results of behaviour of PWD and impact of invoking the RL+LA countermeasure.

5.3 Experimental Methodology

A range of empirical experiments were conducted by using the test network and tools described. These experiments investigated the effect of two key variables:

- ***The proportion of hosts in the network, which are vulnerable to infection (i.e. are running the PWD):*** Values of 25%, 20%, 15% and 10% were investigated. These population values were selected to investigate the effectiveness of the RL+LA scheme as an initial proof of concept on a small scale.
- ***The level of countermeasure implemented:*** A series of experiments were conducted with following settings:
  - No countermeasure (to provide a base-line)
  - Only the local rate limiting from infected hosts (RL only)

- Rate limiting and the alerting protocol implemented with reducing threshold (RL+LA).

For all the experimental tests, N was set to 15 datagrams in 5 seconds, and the counter in Table B: Counters of figure 3.2 was decremented every 30 seconds. These values were selected as a as an initial proof of concept to achieve maximum countermeasure effect on a small scale. Each time, the experiment was started by infecting the same host at IP address 192.168.0.10 with same random number generator seed value while employing different seed value for

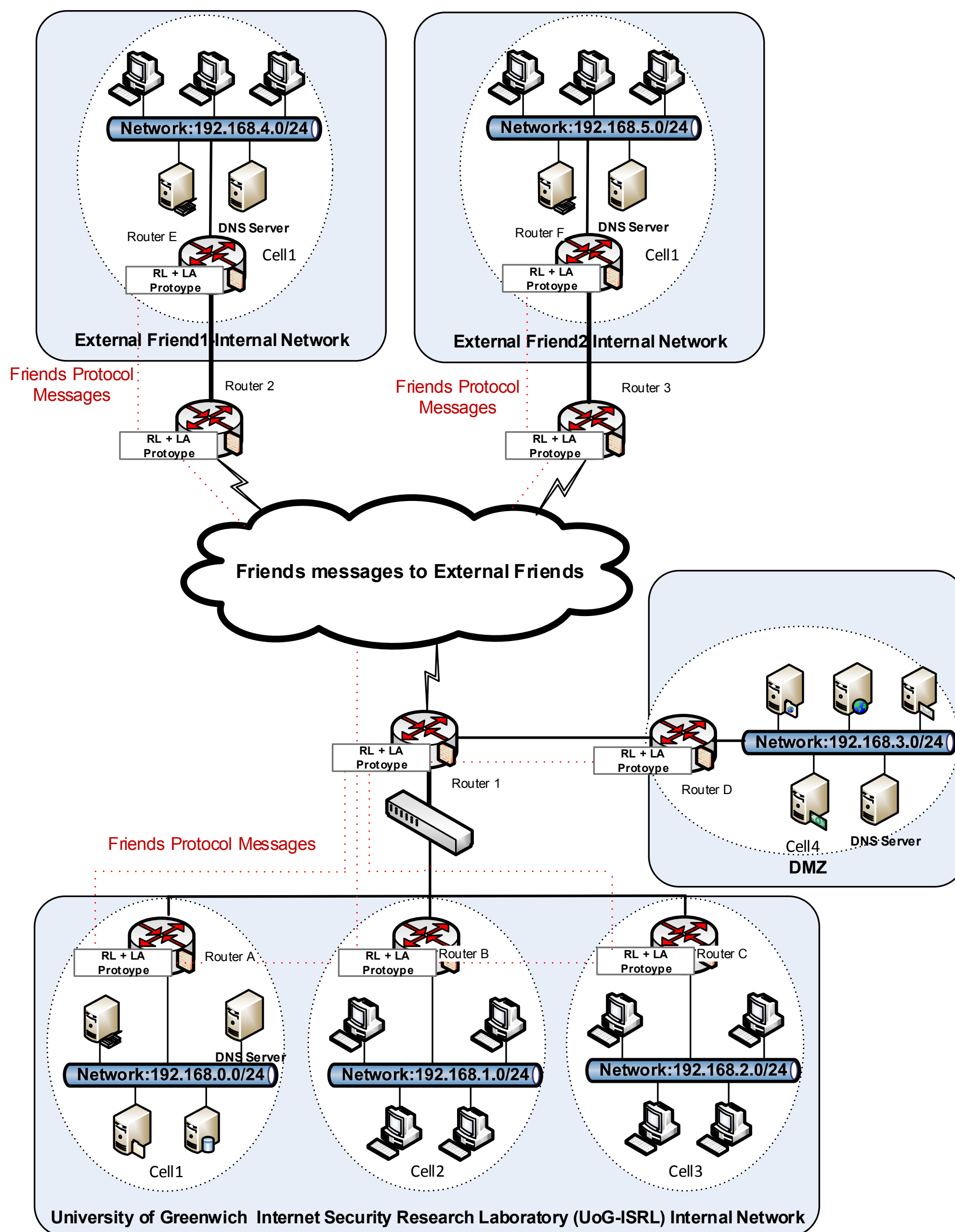

Figure 5-1 Experimental Test Network

all subsequent infected hosts. Each worm infected host is capable of generating 10 UDP datagrams in 2.5 seconds, before it stops, choosing pseudo random destination IP addresses in the pool of the 6 class C networks (192.168.0.0/24 to 192.168. 5.0/24) on port 1434. These pseudo-worm parameters were selected due to the size of experimental networks.

5.4 Experimental Results

Figure 5.2 shows the results of experiments conducted with 25% of hosts vulnerable to infection. Without any protection mechanism in place, all vulnerable hosts are infected within approximately 18 seconds. In the second experiment, with rate limiting only as the countermeasure (no alert messages between peers), 91% (349) of vulnerable hosts are infected within approximately 17 seconds. In the third experiment, rate limiting was implemented with alert messages and 63% (242) of vulnerable hosts were infected, again in around 17 seconds.

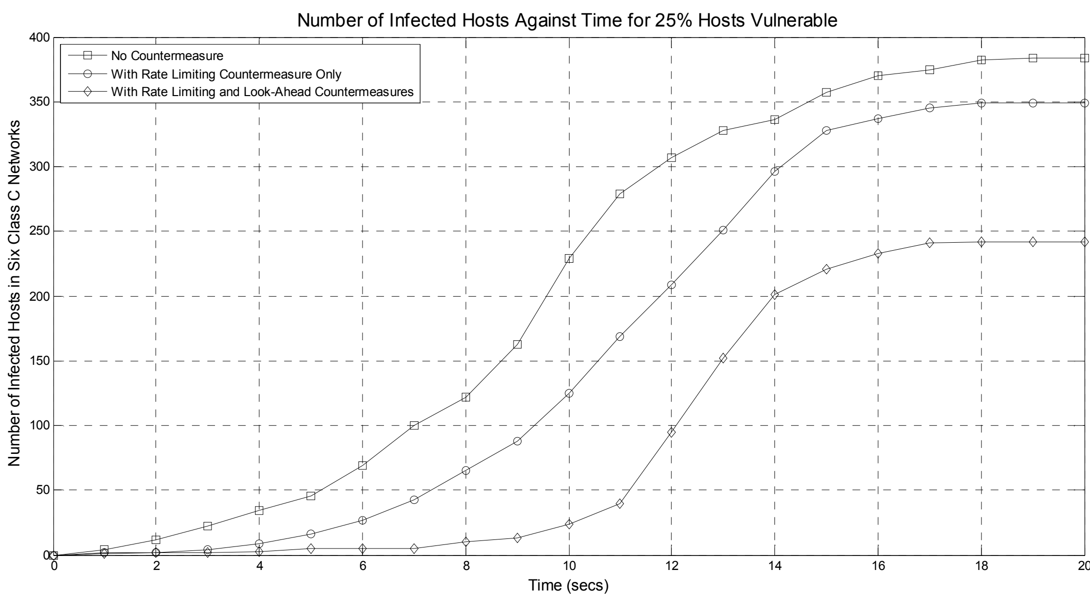

Figure 5-2 Experimental Results with 25 % of Hosts Vulnerable to Infection

Figure 5.3 shows the results of the experiments conducted with 20% of hosts vulnerable. Without any countermeasures in place, all vulnerable hosts were infected within around 18 seconds. In the second experiment, with rate limiting only as the countermeasure (no alert messages between peers), 88% (271) of vulnerable hosts were infected in approximately 20 seconds. In the third experiment, again, rate limiting was implemented with alert messages and, 60% (185) of vulnerable hosts were infected in around 20 seconds.

Figure 5.4 shows the results of the experiments conducted with 15% of the network hosts vulnerable to infection. Without any countermeasures in place, all vulnerable hosts (231) were infected within approximately 17 seconds. In the second experiment, with rate limiting only as the countermeasure (no alert messages between peers), 87.5% (271) vulnerable hosts were infected in approximately 17 seconds. In the third experiment, again, rate limiting was implemented with alert messages and, 56.5% (131) of vulnerable hosts were infected.

Figure 5.5 shows the results of the set of experiments conducted with 10 % of the network hosts vulnerable to infection. Without any countermeasures in place, 56.209% (81) of vulnerable hosts were infected within approximately 35 seconds. In the second experiment, with rate limiting only as the countermeasure (no alert messages between peers), 5.228% (8) vulnerable hosts were infected in approximately 15 seconds. In the third experiment, again, rate limiting was implemented with alert messages and, 2.614% (4) of vulnerable hosts were infected in 7 seconds. It is also noted that multiple runs were required to start the infection process due to the smaller number of vulnerable hosts and as one instance of infected PWD only generated 10 UDP datagrams before it stopped. Hence, experimental results with 10 % of hosts vulnerable to infection are not analyzed further

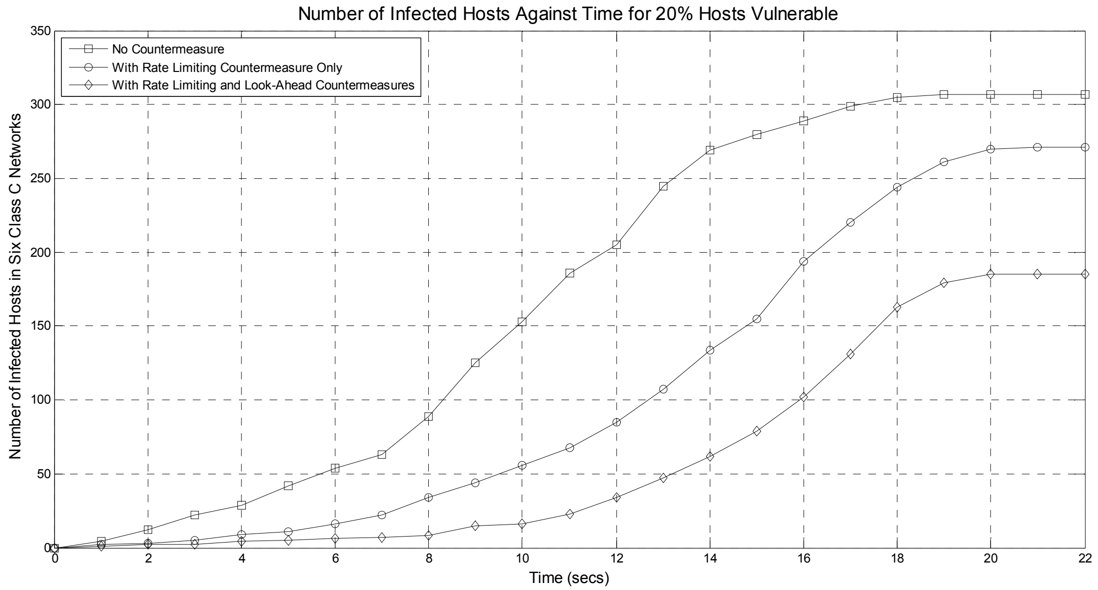

Figure 5-3 Experimental Results with 20 % of Hosts Vulnerable to Infection

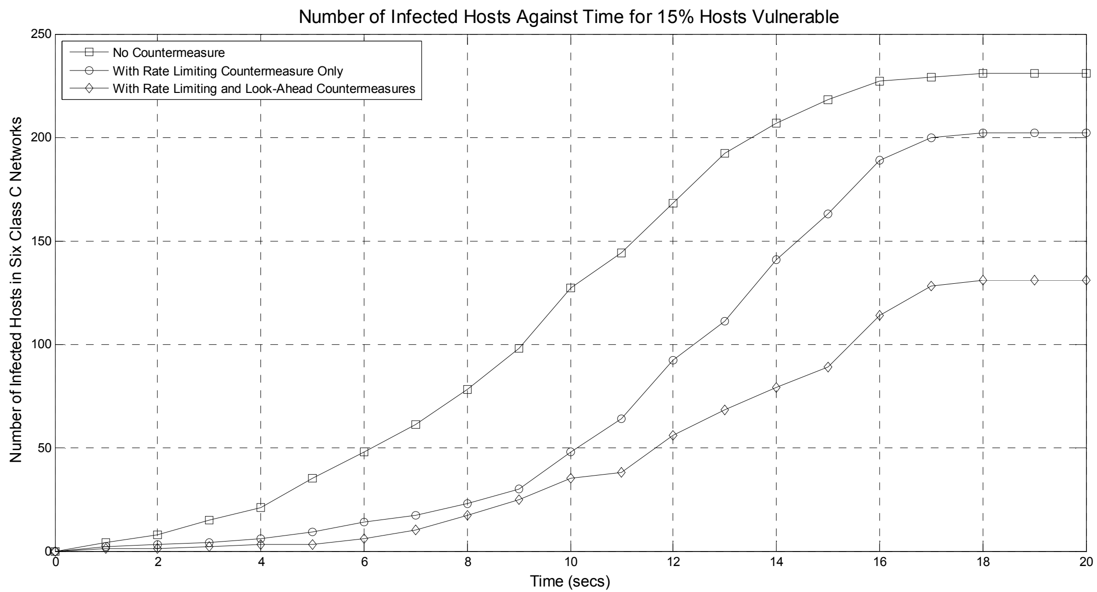

Figure 5-4 Experimental Results with 15 % of Hosts Vulnerable to Infection

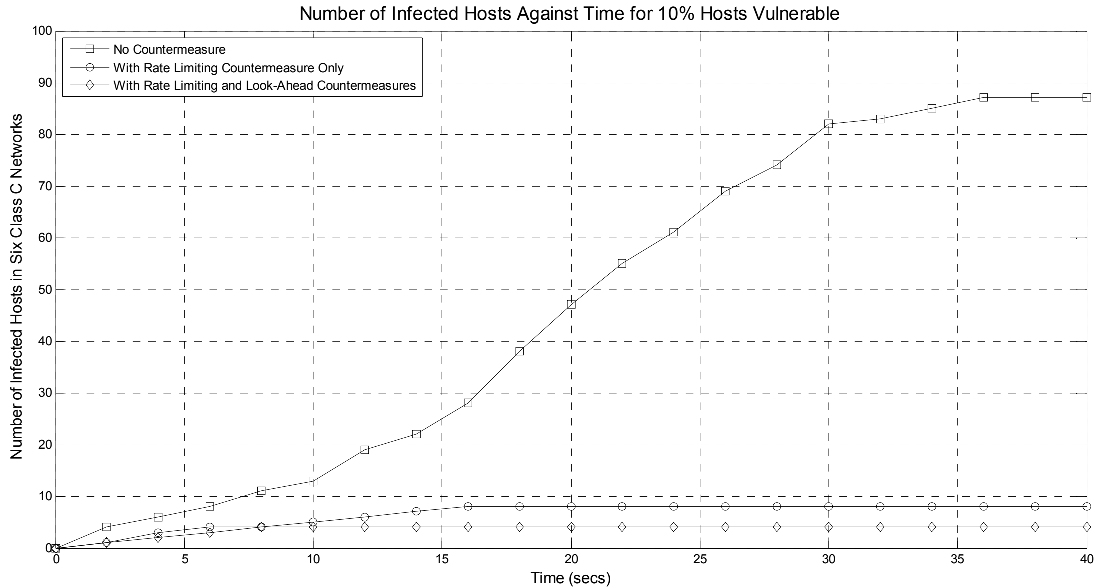

Figure 5-5 Experimental Results with 10 % of Hosts Vulnerable to Infection

5.5 Discussion and Future Work

In all four sets of experiments, it can be seen that rate limiting alone reduces the speed of propagation of the worm as well as the number of hosts ultimately infected. When the Friends protocol messages and the threshold reduction were also implemented, the speed of propagating and the number of hosts infected were further reduced.

Table 5.1 summarizes the results of the complete set of experiments as follows:

- In the first scenario (25% of hosts vulnerable to infection), RL+LA reduces the number of infected hosts to 63% as compared to the RL with 91%.
- In the second scenario (20% of hosts vulnerable to infection), RL+LA reduces the number of infected hosts to 60% as compared to the RL only with 88%.
- In the third scenario (15% of hosts vulnerable to infection), RL+LA reduces the number of infected hosts to 56.6 % as compared to the RL only with 87.5%.

Table 5-1 Summary of Initial Results

Susceptible Population	Infected Population		
	No Countermeasures	RL Countermeasure N=15, t=5	RL+LA Countermeasure Threshold I N=15, t=5, K=30 reducing N to half
25 % (384 of 1536 Hosts)	100 % (384 Hosts)	91 % (349 Hosts)	63 % (242 Hosts)
20 % (307 of 1536 Hosts)	100 % (307 Hosts)	88 % (271 Hosts)	60 % (185 Hosts)
15 % (231 of 1536 Hosts)	100 % (231 Hosts)	87.5 % (202 Hosts)	56.5 % (131 Hosts)
10 % (153 of 1536 Hosts)	56.209% (81 Hosts)	5.228 % (8 Hosts)	2.614 % (4 Hosts)

- In the last scenario (10% of hosts vulnerable to infection), the number of hosts ultimately infected with RL+LA is 2.164% as compared to the RL only with 5.228%.

Figure 5.6 shows the % ssusceptible hosts infected for experimental test 1-9. From this set of experiments, it is clearly observed that the lower the percentage of susceptible population of worm, the more effective is the countermeasure.

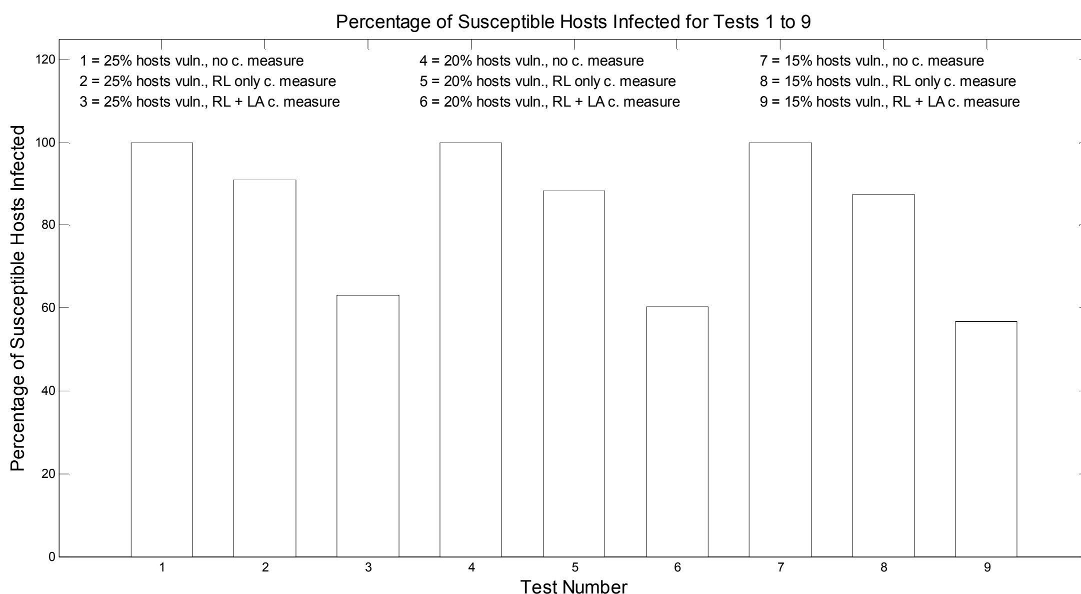

Figure 5-6 % of Susceptible Hosts Infected for Experimental Tests 1-9

Furthermore, it is empirically observed from the set of experiments without any countermeasure, that infection process of PWD follows the s-shaped curve pattern (Moore et al., 2003).

In terms of future work, the performance of the RL+LA scheme with different threshold values needs to be explored on a large scale network, with real worm conditions, which will be reported in the next chapter.

5.6 Chapter Summary

This chapter has presented the initial set of results of launching a pseudo random scanning worm (PWD), on a small scale in an isolated experimental testbed (VMT) and invoking a worm detection and containment scheme, RL+LA. It has also reported the results of invoking rate limiting (RL) countermeasure and its comparison with RL+LA. From this chapter, it is concluded that RL+LA is an effective approach for worm detection and containment. The next chapter will report the results of launching countermeasures such as RL and RL+LA on large network by employing more realistic test conditions.

6 EXPERIMENTAL RESULTS FOR THE RL+LA SCHEME ON CLASS A SCALE NETWORKS WITH REAL WORM OUTBREAK ATTRIBUTES

6.1 Introduction

Chapter 5 presented the results and discussion of an initial set of experiments conducted with countermeasures (RL and RL+LA) on a small scale. The results showed that the RL+LA countermeasure performed significantly better in comparison to the RL only countermeasure on a small scale with six class C networks and with the limited scanning rate of Pseudo-Worm Daemon (PWD). The thesis now considers how this work is comparable to worms with real worm outbreak attributes on class A scale networks. Hence there is a need to empirically analyse the propagation of different random scanning worms, and to investigate the impact of the designed countermeasure (RL+LA) by using the attributes of real worm outbreaks such as SQL Slammer and Witty on a class A scale networks such as (class A network with address space of 16 million hosts). Therefore, a detailed set of experimental work has been conducted to analyse the effectiveness of the RL+LA countermeasure. This chapter presents the results and discussion of these experiments.

6.1.1 Chapter Layout

This chapter begins with the background to the experiments in section 6.1. Section 6.2 details the SQL Slammer outbreak characteristics, experimental setup and experimental methodology, used to conduct these experiments, and experimental results of Pseudo-Slammer worm experiments; while section 6.3 presents the Witty outbreak characteristics, experimental setup and experimental methodology, used to conduct these experiments, and experimental results of Pseudo-Witty worm experiments. Section 6.4 presents the detailed discussion of the experimental results by discussing the impact of implementing different countermeasures (RL and RL+LA), impact of network properties to RL and RL+LA countermeasure experimental test results, RL+LA countermeasure overhead, and applicability of RL and RL+LA Class A experimental test results to the Internet scale. Finally, a summary of this

chapter is presented in section 6.5.

6.2 Pseudo-Slammer Worm Experiments

6.2.1 Slammer Worm

Moore et al (Moore et al., 2003) reported some key characteristics of the Slammer outbreak of 2003 which can be summarised as follows:

- 18 hosts per million of the entire IPv4 address space were susceptible to infection.
- The maximum recorded scanning rate of Slammer was 26,000 datagrams per infected host per second. This figure seems reasonable while considering the upper bound of 100BaseT interface and the worm Ethernet frame size of 430 bytes.
- The average scanning rate of Slammer was 4000 datagrams per worm instance per second during its entire infection period.

6.2.2 Experimental Setup

In order to empirically analyse the behaviour of the Slammer worm and to validate the RL+LA prototype on a class A scale, an experimental test network was configured on the Virtualized Malware Testbed (VMT) (reported in Chapter 4 of this thesis), comprising of a single Class A address space 10.0.0.0/8 but divided into four subnets; 10.0.0.0/10, 10.64.0.0/10, 10.128.0.0/10 and 10.192.0.0/10 as shown in Figure 6.1. These four subnets were connected through a central router by using RIP, configured on Quagga. Eight further Quagga based routers were implemented (two for each subnet). The RL+LA prototype was installed on each of these eight routers. The RL+LA prototype installed on routers A,B,C and D was configured to rate limit the outbound connection based on DNS anomalies and to send Friends protocol messages whereas the RL+LA prototype installed on the border routers only forwarded the Friends protocol messages received from internal and external friends. One Linux based virtual host was running in each subnet to provide a DHCP service and logging service for the PWD. DSL was installed with the PWD on each of the susceptible virtualised hosts. All hosts in the network are time synchronized by using Network Time Protocol (NTP).

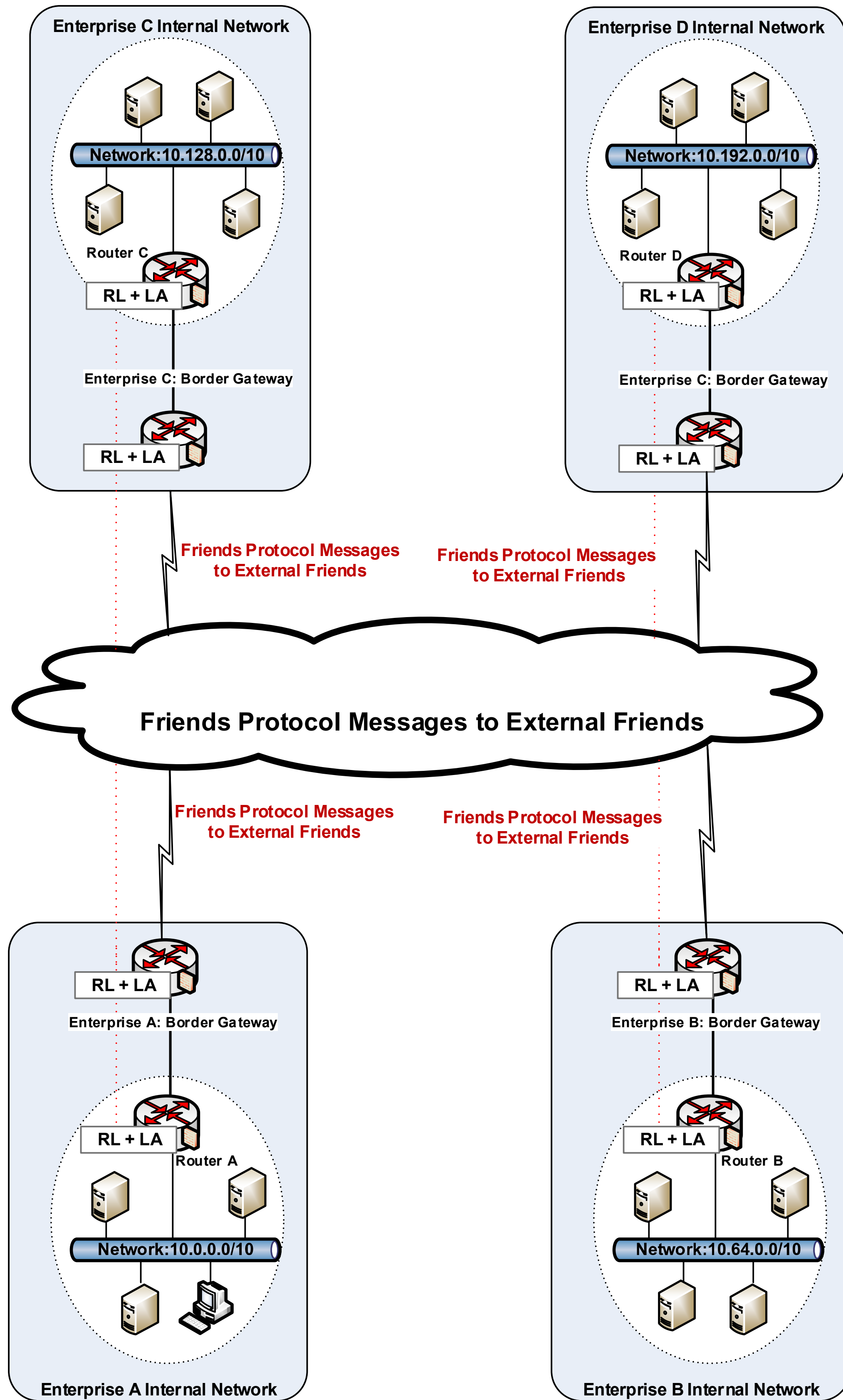

Figure 6-1 Slammer Worm Experimental Test Network

6.2.3 Experimental Methodology

As reported in section 6.2.1, approximately 18 hosts per million of the entire IPv4 addresses space were susceptible to infection with Slammer and it achieved an average scan rate of 4,000 datagrams per infected host per second.

A single class A network has 2^{24} hosts, and so will contain 2^{24} * 18/1,000,000 = 302 susceptible hosts. On this basis, 302 virtual hosts with the Slammer like pseudo-worm daemon were deployed across the four subnets. Each worm daemon was configured to scan within a single class A network (10.0.0.0/8). In order to avoid overloading the server farm hardware (in which case the experiments would have been measuring the effect of the hardware restrictions, rather than the properties of the worm), the average worm scanning rate was scaled down by a factor of 80 (scaling factor of 80 was chosen due to resource limitations on DSL based virtualized hosts) . Therefore, based on an average scan rate reported by Moore et al of 4000 scans per second, the Pseudo-Slammer network daemon was configured to scan at 50 scans per second in the set of Slammer experiments.

Four series of experiments were conducted:

- With no countermeasure (to provide a base-line),
- With only the local rate limiting from infected hosts,
- With both rate limiting and the alerting protocol implemented with reducing threshold to half.
- With both rate limiting and the alerting protocol implemented with further reducing threshold to approximately 27 % of the original value.

Each time, the experiment was started by infecting the same host with same random number generator seed value while employing different seed value for all subsequent infected hosts. The threshold values N in time t and counter in Table B: Counters of figure 3.2 were also scaled up by a factor of 80. For the purpose of clarity, the un-scaled values will be used henceforth in this thesis.

6.2.4 Experimental Results

6.2.4.1 No Countermeasure

Figure 6.2 shows the results of a set of three experiments conducted without implementing any countermeasures. In the first experiment, all 302 susceptible hosts were infected in 15.07 minutes. In the seconds experiment, all 302 susceptible hosts were infected in 14.58 minutes. While in the third experiment, all 302 susceptible hosts were infected in 14.45 minutes.

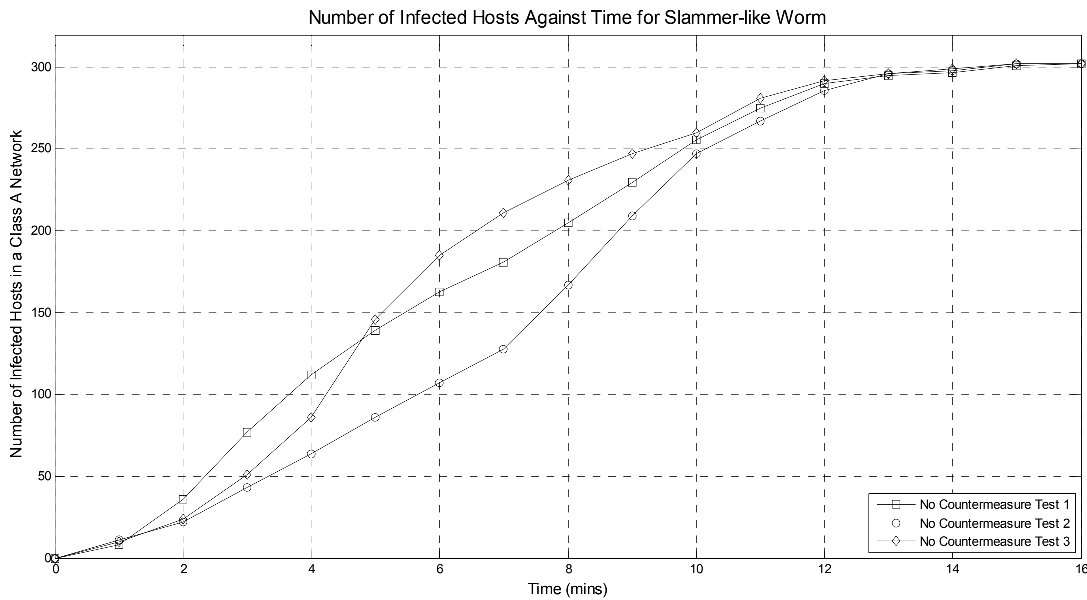

Figure 6-2 Experimental Results of Pseudo-Slammer Worm with No Countermeasures

6.2.4.2 RL Countermeasure

Figure 6.3 shows the results of a set of three experiments conducted with the RL countermeasure (local rate limiting from infected hosts). For all three tests, N was set to 15 datagrams in 5 seconds, and the counter in Table B: Counters of figure 3.2 being decremented every 30 seconds. In the first experiment, all 302 susceptible hosts were infected in 41.18 minutes. In the second experiment, all 302 susceptible hosts were infected in 40.39 minutes. While in the third experiment, all 302 susceptible hosts were infected in 41.24 minutes.

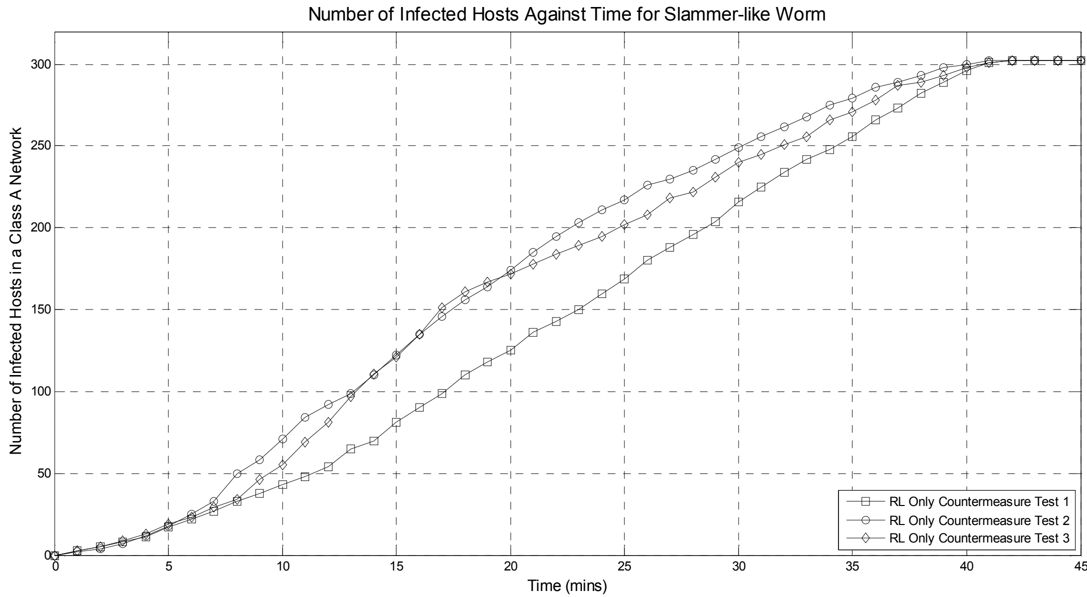

Figure 6-3 Experimental Results of Pseudo-Slammer Worm with RL Countermeasure

6.2.4.3 RL+LA Countermeasure

Figure 6.4 shows the results of a set of three experiments conducted with the RL+LA countermeasure (with both rate limiting and the alerting protocol implemented with reducing threshold to half). For all three tests, N was set to 15 datagrams in 5 seconds, and the counter in Table B: Counters of figure 3.2 being decremented every 30 seconds. In the first experiment, all 302 susceptible hosts were infected in 64.54 minutes. In the second experiment, all 302 susceptible hosts were infected in 66.24 minutes. While in the third experiment, all 302 susceptible hosts were infected in 64.36 minutes.

Figure 6.5 shows the results of a set of three experiments conducted with RL+LA countermeasures (with both rate limiting and the alerting protocol implemented with reducing threshold to half). For all three tests, N was set to 8 datagrams 5 seconds, and the counter in Table B: Counters of figure 3.2 being decremented every 30 seconds. In the first experiment, all 302 susceptible hosts were infected in 96.37 minutes. In the seconds experiment, all 302 susceptible hosts were infected in 101.38 minutes. While in the third experiment, all 302 susceptible hosts were infected in 98.58 minutes.

6. EXPERIMENTAL RESULTS FOR THE RL+LA SCHEME ON CLASS A SCALE NETWORKS WITH REAL WORM OUTBREAK ATTRIBUTES

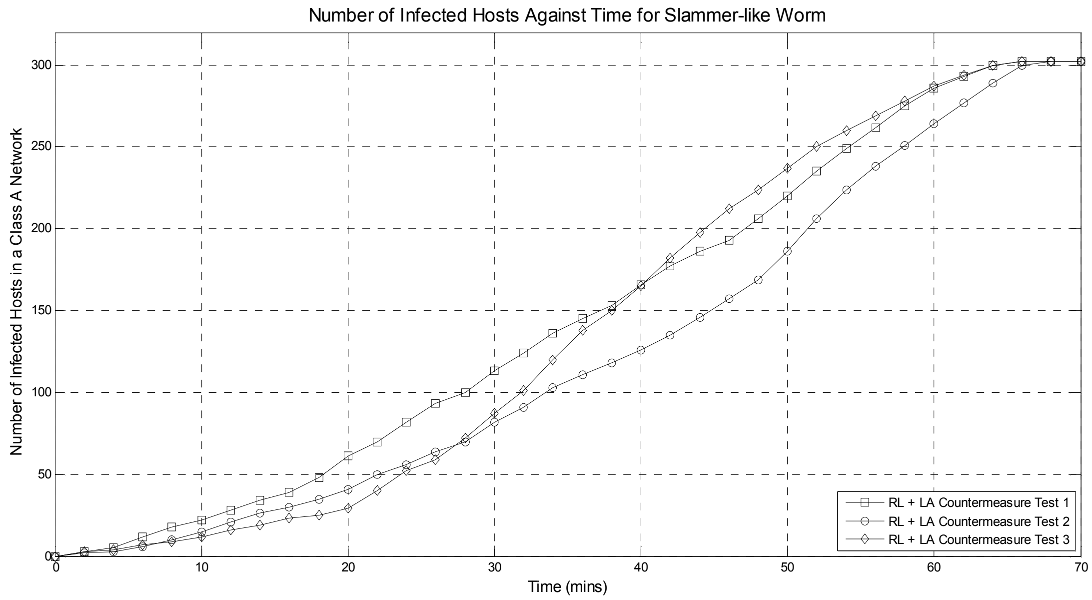

Figure 6-4 Results of Pseudo-Slammer Worm with RL+LA Countermeasure Threshold I

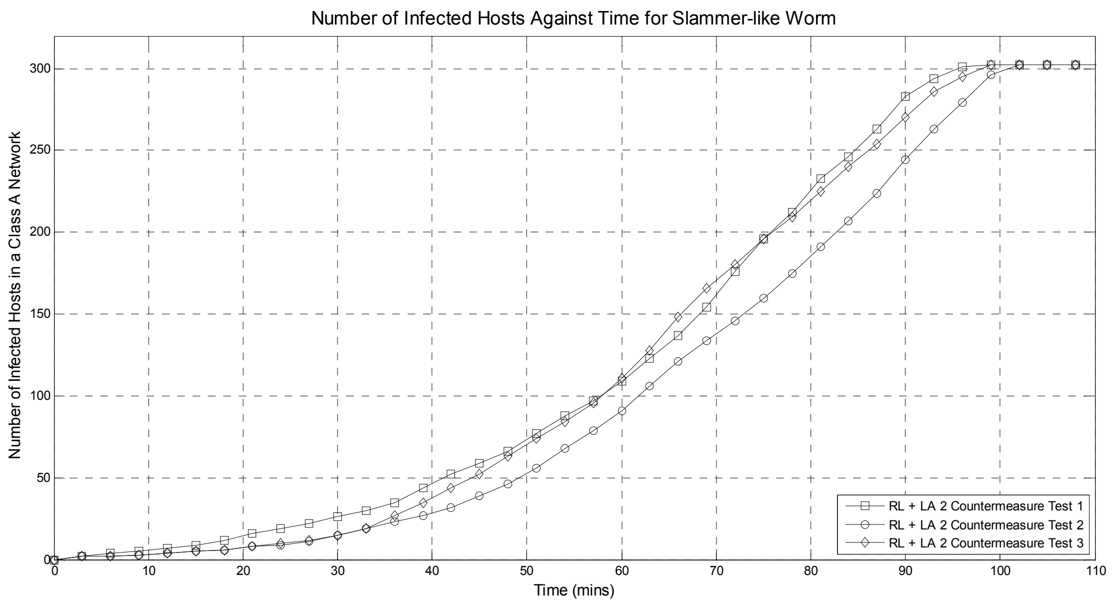

Figure 6-5 Results of Pseudo-Slammer Worm with RL+LA Countermeasure Threshold II

From these four sets of experiments, following key points have been observed in all three Pseudo-Slammer worm experiments:

- Firstly, the infection process in all three Pseudo-Slammer worm experiments with No Countermeasure in figure 6.2 approximates to standard s-shaped curve (Moore et al., 2003).
- Secondly, in the middle part of outbreaks of all the Pseudo-Slammer worm experiments, the experimental curves diverge significantly, due to statistical variations while in the last stage of the experiments, the experimental curve form plateau to finish the infection process.

6.3 Pseudo-Witty Worm Experiments

6.3.1 Witty Worm

Shannon et al. (Shannon and Moore, 2004) reported some key characteristics of the Witty worm outbreak of 2004 which can be summarised as follows:

- The Susceptible population of the Witty worm was 12, 000 or between 2 and 3 hosts per million of the entire IPV4 address space.
- Witty worm had a variable datagram size, with an Ethernet frame size between 796 and 1307 bytes.
- The average scanning rate of Witty was 357 datagrams per infected host per second during its entire infection period while the maximum recorded scanning rate was 970 datagrams per host per second.
- Witty also utilized an initial hit-list of 110 hosts which were reported to have been infected in the first 10 seconds of launch. Of these110 hosts, 38 hosts were transferring 9700 datagrams per host per second continuously for a period of an hour.

6.3.2 Experimental Setup

In order to empirically analyse the behaviour of the Witty worm and to validate the RL+LA prototype on a class A scale network, an experimental test network was configured on the Virtualized Malware Testbed (reported in Chapter 5 of this thesis), comprising of a two Class A address space 10.0.0.0/8 and 11.0.0.0/8 but divided into four subnets; 10.0.0.0/10, 10.128.0.0/9, 11.0.0.0/9 and 10.128.0.0/9 as shown in Figure 6.6. All the other network

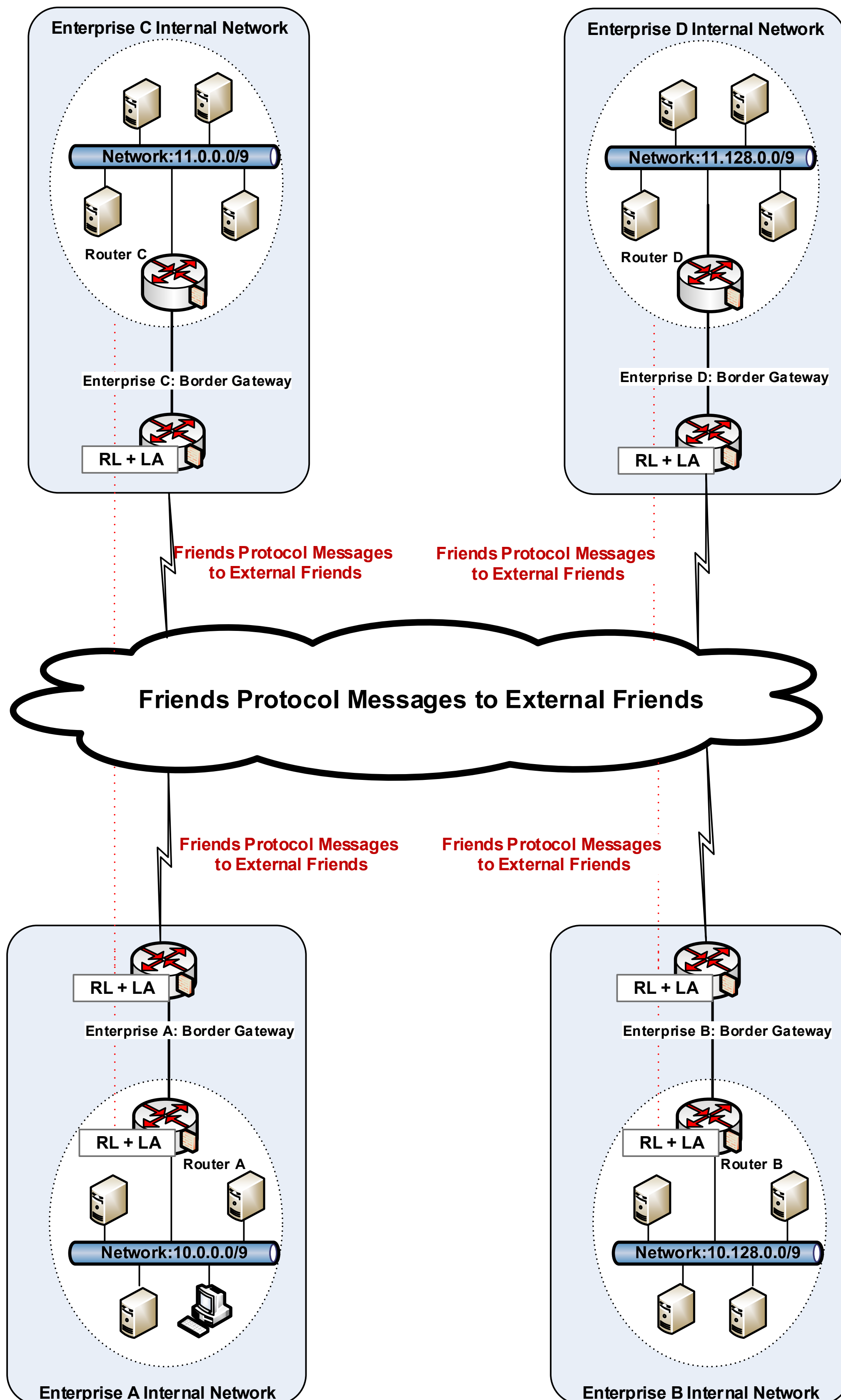

Figure 6-6 Witty Worm Experimental Test Network

elements of experimental test network were the same as those defined previously in section 6.2.1.

6.3.3 Experimental Methodology

As reported in section 6.3.1, Witty had 3 hosts per million of the entire IPv4 addresses space were susceptible to infection with an average scan rate of 357 datagrams per infected host per second.

A single class A network has 2^{24} hosts, and so 2 class A networks will contain 2^{24} * 2(3/1,000,000) = 101 susceptible hosts. On this basis, 101virtual hosts with the Witty like pseudo-worm daemon were deployed across the four subnets. Each worm daemon was configured to scan within two class A networks (10.0.0.0/8, 11.0.0.0/8) at a scanning rate of 357 scans per host per second while using an initial hit-list of one susceptible host held by the first infected host.

Three series of experiments were conducted:

- With no countermeasure (to provide a base-line),
- With only the local rate limiting from infected hosts,
- With both rate limiting and the alerting protocol implemented with reducing threshold to half.

Each time, the experiment was started by infecting the same host with same random number generator seed value while employing different seed value for all subsequent infected hosts.

6.3.4 Experimental Results

6.3.4.1 No Countermeasure

Figure 6.7 shows the results of a set of three experiments conducted without implementing any countermeasures by utilizing the Pseudo-Witty worm. In the first experiment, all 101 susceptible hosts were infected in 96.22 minutes. In the second experiment, all 101 susceptible hosts were infected in 95.16 minutes. While in the third experiment, all 101 susceptible hosts were infected in 94.46 minutes.

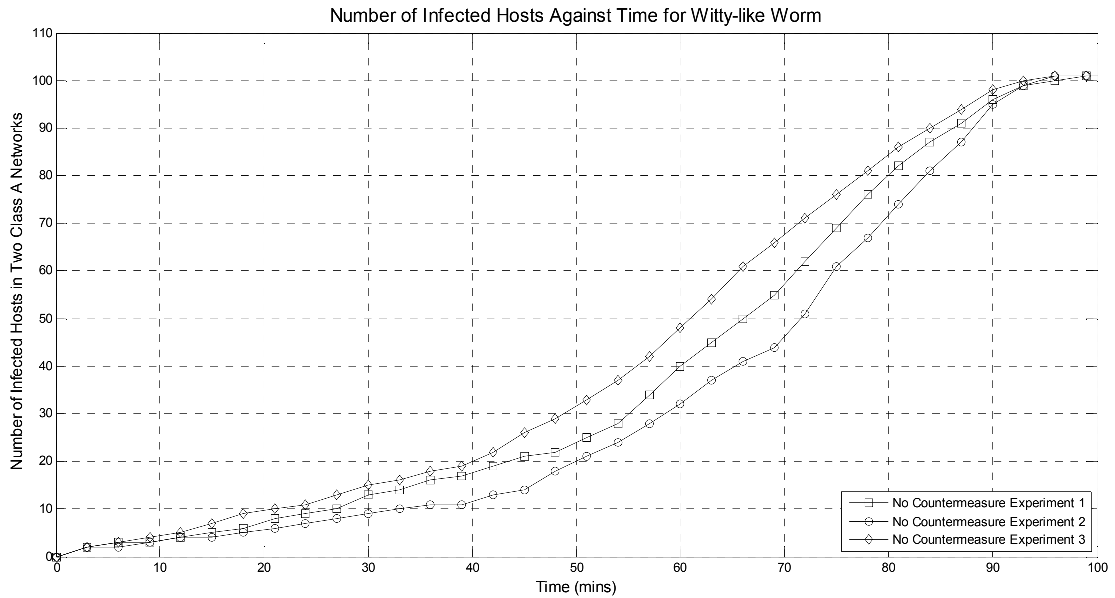

Figure 6-7 Results of Pseudo-Witty Worm

6.3.4.2 RL Countermeasure

Figure 6.8 shows the results of a set of three experiments conducted with the RL countermeasure (local rate limiting from infected hosts). For all three tests, N was set to 15 datagrams in 5 seconds, and the counter in Table B: Counters of figure 3.2 was decremented every 30 seconds without any scaling factor. In the first experiment, all 101 susceptible hosts were infected in 312.21 minutes. In the second experiment, all 101 susceptible hosts were infected in 309.56 minutes. While in the third experiment, all 101 susceptible hosts were infected in 307.08 minutes.

6.3.4.3 RL+LA Countermeasure

Figure 6.9 shows the results of a set of three experiments conducted with the RL+LA countermeasure (with both rate limiting and the alerting protocol implemented by reducing the threshold to half). For all three tests, N was set to 15 datagrams in 5 seconds, and the counter in Table B: Counters of figure 3.2 was decremented every 30 seconds without any scaling factor. In the first experiment, all 101 susceptible hosts were infected in 562.10 minutes. In the second experiment, all 101 susceptible hosts were infected in 571.51 minutes. While in the third experiment, all 101 susceptible hosts were infected in 585.50 minutes.

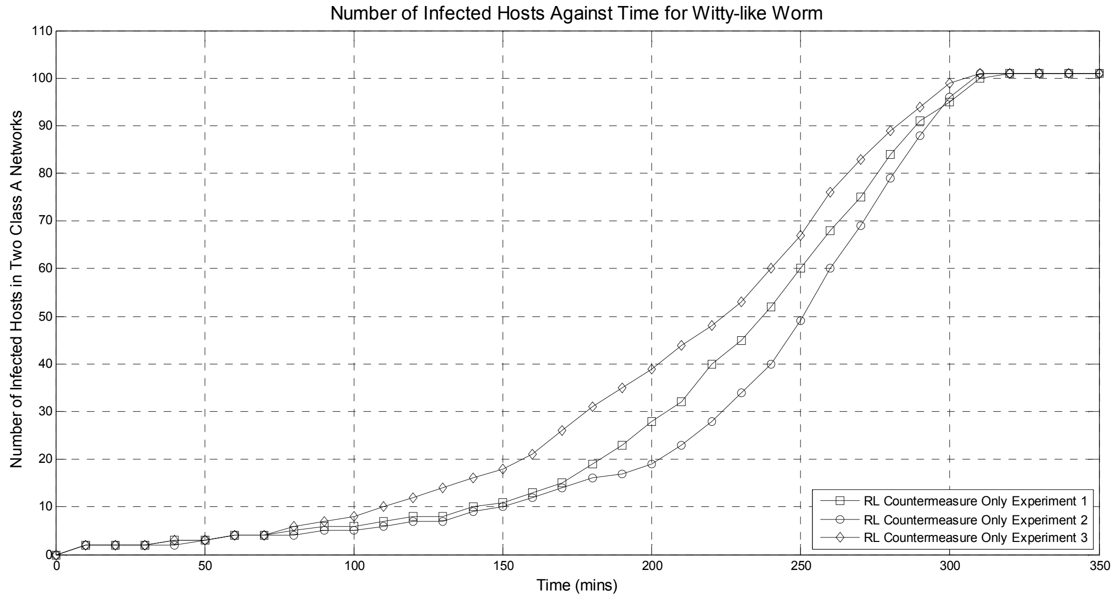

Figure 6-8 Results of Pseudo-Witty Worm with RL Countermeasure

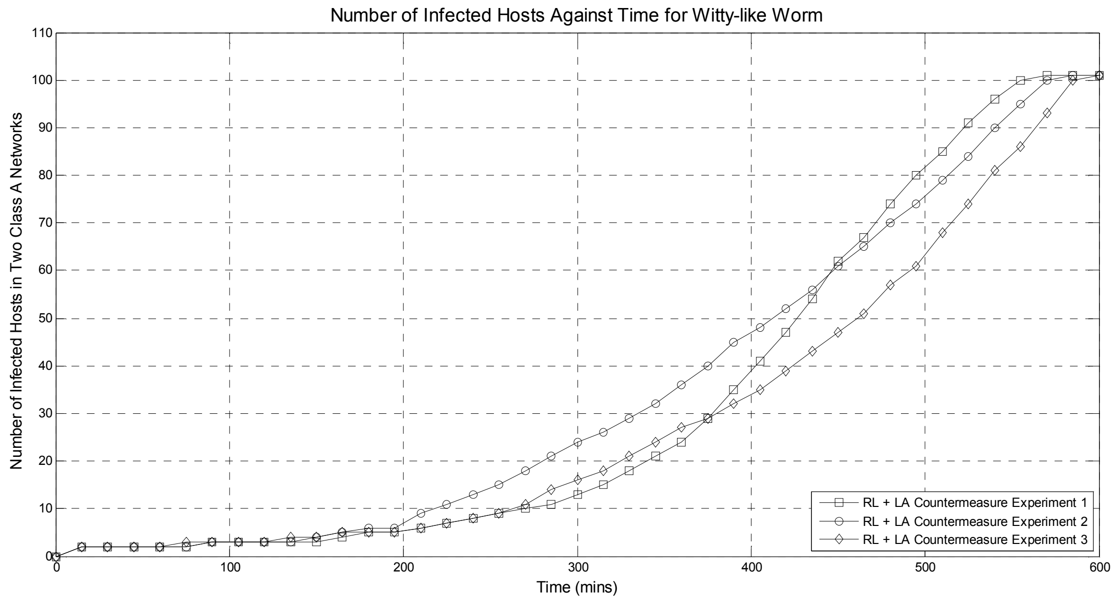

Figure 6-9 Results of Pseudo-Witty Worm with RL+LA Countermeasure

From these three sets of experiments, following key points have been observed in all three Pseudo-Witty worm experiments:

- Firstly, the infection process in all three Pseudo-Witty worm experiments with No Countermeasure in figure 6.7 approximates to standard s-shaped curve (Moore et al., 2003).
- Secondly, in the middle part of outbreaks of all the Pseudo-Witty worm experiments, the experimental curves diverge significantly, due to statistical variations while in the last stage of the experiments, the experimental curve form plateau to finish the infection process.

6.4 Discussion

6.4.1 Comparison of Pseudo-Slammer Worm Results

Table 6.1 summarizes all of the experiments conducted by using the Pseudo-Slammer worm with No countermeasures, with the RL countermeasure and the RL+LA countermeasure with different 2 different thresholds and average of each of three set of experiments.

Table 6-1 Results of Pseudo-Slammer Worm

Results of Pseudo-Slammer Worm				
	No Countermeasure	**RL Countermeasure N=15, t=5**	**RL+LA Countermeasure Threshold I N=15, t=5, K=30 reducing N to half**	**RL+LA Countermeasure Threshold II N=8, t=5, K=30 reducing N to half**
Experiment 1	15.07 min	41.18 min	64.54 min	96.37 min
Experiment 2	14.58 min	40.39 min	66.24 min	101.38 min
Experiment 3	14.45 min	41.24 min	64.36 min	98.58 min

Figure 6.12 shows the average of these results for all four set of experiments. In the first scenario (No Countermeasure), all susceptible hosts were infected in 14.45 minutes. In the second scenario (RL countermeasure), all susceptible hosts were infected in 41.07 min minutes. In the third scenario (RL+LA countermeasure Threshold I), all susceptible host were infected in 65.18

minutes. In the fourth scenario (RL+LA countermeasure threshold II), all susceptible host were infected in 99.07 minutes.

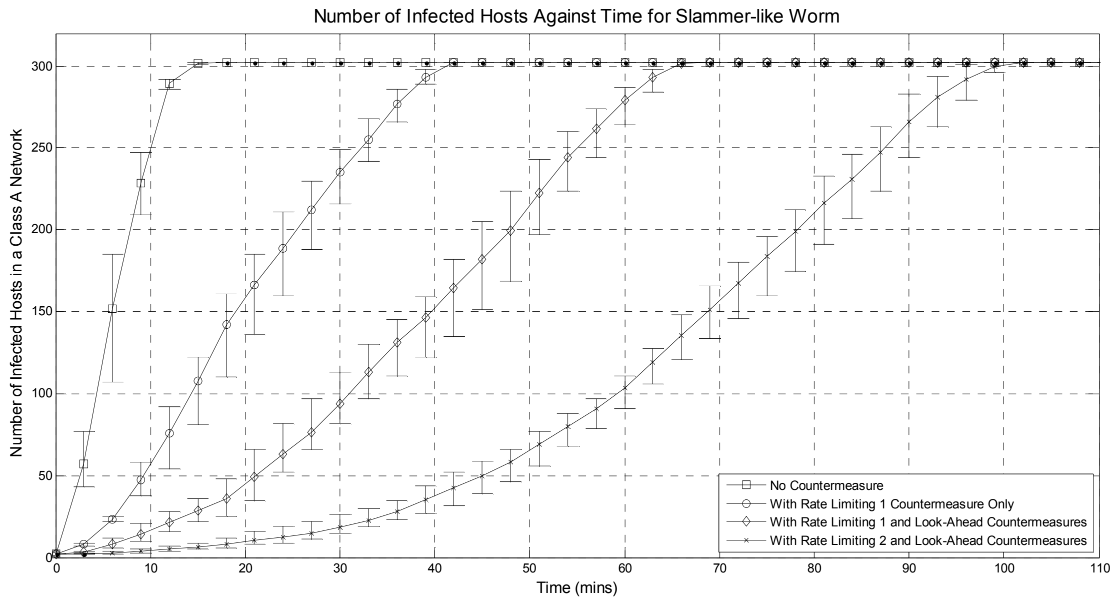

Figure 6-10 Comparison of Pseudo-Slammer Worm Results

Figure 6.13 shows the average time t, of these results for all four set of Pseudo-Slammer worm experiments. It can be seen that values of *t* increase as countermeasures are implemented; showing RL+LA with Threshold II is most effective.

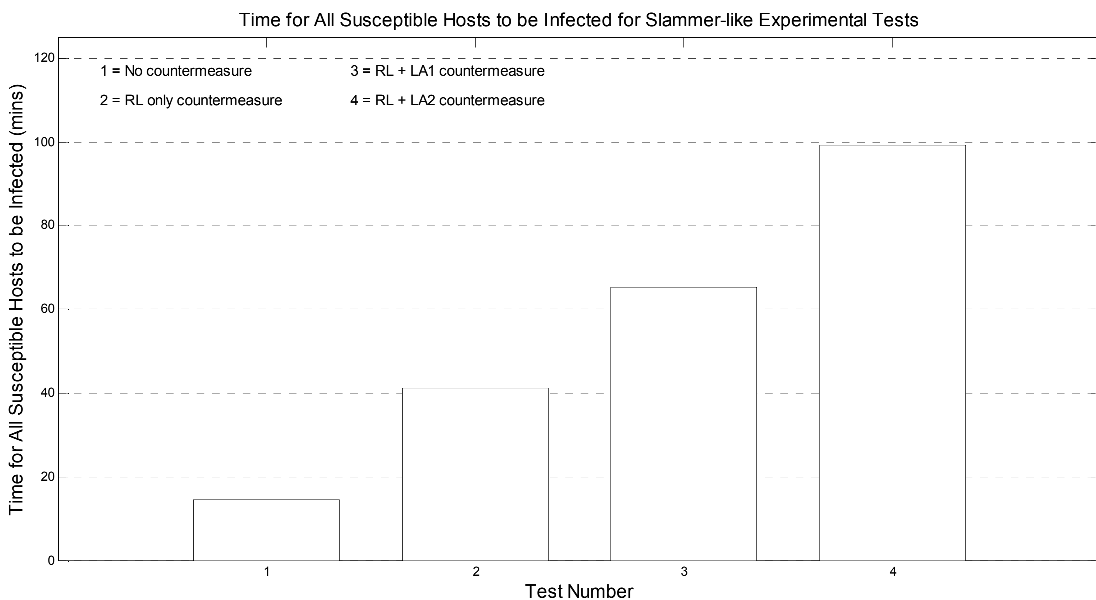

Figure 6-11 Time of Infection for Pseudo-Slammer Experimental Tests

From these results, the following points were observed:

- The RL countermeasure decreases the propagation rate of the Pseudo-Slammer worm and increases the time required to infect all susceptible hosts by 2.8 times in comparison to no countermeasure. But, ultimately, all susceptible hosts were infected. Hence RL (rate limiting) cannot stop the spread of Pseudo-Slammer worm.
- The RL+LA countermeasure with Threshold I (N=15, t=5, K=30) further decreases the propagation rate of the Pseudo-Slammer worm and increases the time required to infect all susceptible hosts by 4.5 times in comparison to no countermeasure. But, ultimately, all susceptible hosts were infected.
- The RL+LA countermeasure with Threshold II (N=8, t=5, K=30) further decreases the propagation rate of the Pseudo-Slammer worm and increases the time required to infect all susceptible hosts by 6.8 times in comparison to no countermeasure. But, finally, all susceptible hosts were infected.
- The RL+LA countermeasure with a more restricted threshold performs the best in decreasing the propagation rate of the Pseudo-Slammer worm while increasing the time to worm full infection.
- The RL+LA countermeasure with rate limiting on the gateways (edge routers) itself is not enough to stop the propagation of the Pseudo-Slammer worm, as it stops the infection process at the gateway but the Pseudo-Slammer worm continues to spread in the internal LAN and ultimately infects all the susceptible hosts in that LAN.

6.4.2 Comparison of Pseudo-Witty Worm Results

Table 6.2 summarizes all of the experiments conducted by using the Pseudo-Witty worm with No countermeasures, with the RL countermeasure and the RL+LA countermeasure and average of each of three set of experiments.

Table 6-2 Results of Pseudo-Witty Worm

Results of Pseudo-Witty Worm			
	No Countermeasure	**RL Countermeasure N=15, t=5, K=30**	**RL+LA Countermeasure N=15, t=5, K=30 reducing N to half**
Experiment 1	96.22 min	312.21 min	562.10 min
Experiment 2	95.16 min	309.56 min	571.51 min
Experiment 3	94.46 min	307.08 min	585.50 min

Figure 6.14 shows the average of these results for all three set of experiments (i-e; No countermeasures, the RL countermeasure, the RL+LA countermeasure). In the first scenario (No countermeasure), all susceptible hosts were infected in 95.28 minutes. In the second scenario (With RL countermeasure), all susceptible host were infected in 309.48 minutes. In the third scenario (With RL+LA countermeasure), all susceptible host were infected in 573.18 minutes.

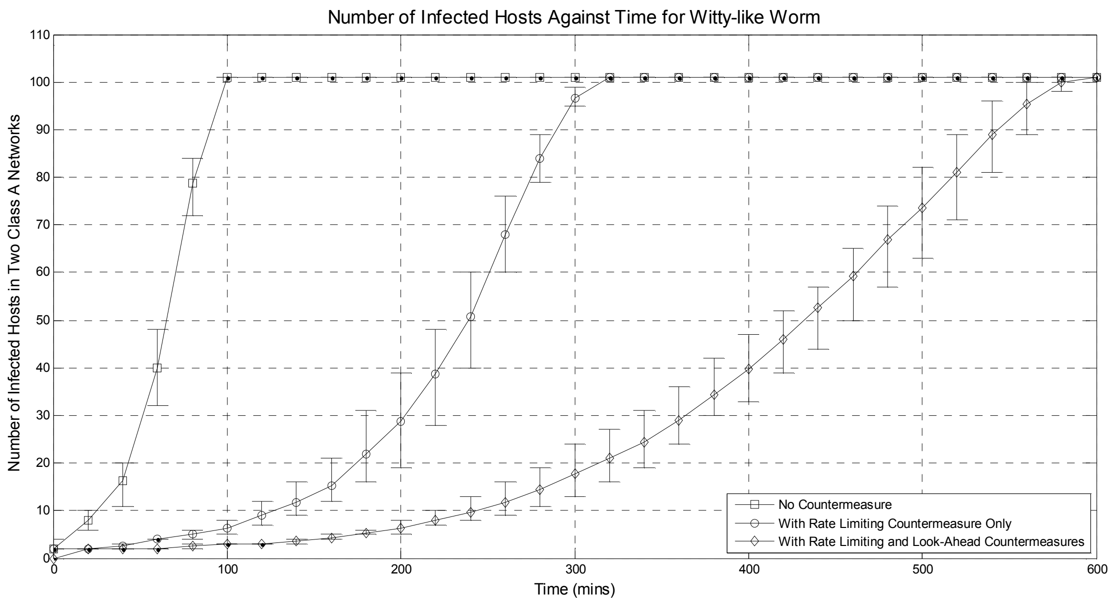

Figure 6-12 Comparison of Pseudo-Witty-Worm Results

Figure 6.15 shows the average time t, of these results for all four set of Pseudo- Witty worm experiments. It can be seen that values of *t* increase as countermeasures are implemented; showing RL+LA countermeasure with Threshold I is most effective.

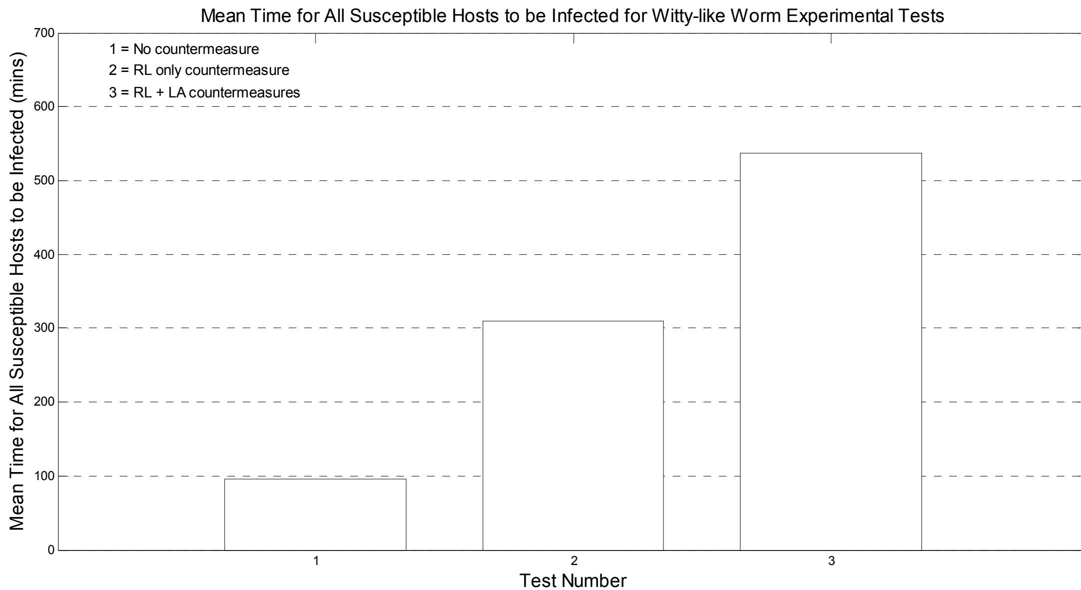

Figure 6-13 Time of Infection for Pseudo-Witty Experimental Tests

From these results, the following points were observed:

- The RL countermeasure decreases the propagation rate of the Pseudo-Witty worm and increases the time required to infect all susceptible hosts by 3.2 times in comparison to no countermeasure. But, ultimately, all hosts were infected. Hence RL (rate limiting) cannot stop the spread of Pseudo-Witty worm.
- The RL+LA countermeasure with Threshold I (N=15, t=5, K=30) further decreases the propagation rate of the Pseudo-Witty worm and increases the time required to infect all susceptible hosts by 6.0 times in comparison to no countermeasure. But, ultimately, all vulnerable hosts were infected.
- The RL+LA countermeasure with rate limiting on the gateways (edge routers) itself is not enough to stop the propagation of the Pseudo-Witty worm, as it stops the infection process at the gateway but the

Pseudo-Witty worm continues to spread in the internal LAN and ultimately infects all the susceptible hosts in that LAN.

6.4.3 Alternative Network Topologies

It should be noted that the experimental results reported relate to the specific network topologies employed for the experiments. In considering the general applicability of the results, the following should be noted:

- More complex network topologies, particularly where WAN links are included will exhibit higher network latency times and so the speed of propagation of both a worm and the Friends Protocol messages would be reduced to some extent, impacting on the performance of the countermeasure.
- The experimental tests were conducted in a network where all of the network segments were protected by the countermeasure scheme. In network topologies where this is not the case, the effectiveness of the countermeasure is likely to be reduced.

6.4.4 RL+LA Countermeasure Overhead

The RL+LA prototype adds memory and processing overhead to the gateway routers on which it is installed, as it stores a copy of DNS A records on the gateway. Hence, extra memory and processing power is needed on the each gateway with RL+LA prototype.

6.4.5 Applicability of RL+LA Experimental Results on the Internet Scale

The In general, the points set out in section 6.4.3 apply to the extrapolation of the experimental results to the scale of the whole internet. In particular, it is likely to be infeasible to deploy the proposed countermeasure scheme to all autonomous systems (ASs) which make up the internet, and so the countermeasure is likely to be less effective in this context.

6.5 Chapter Summary

This chapter has presented the results of launching the Pseudo-Slammer and the Pseudo-Witty worm with specific outbreak conditions on class A scale

networks to evaluate a worm detection and containment scheme, RL+LA. From these results, it is concluded that the outcomes are broadly comparable to the real worm outbreaks, thereby validating the authenticity of the experiments. It has also been observed that the designed countermeasure scheme, RL+LA decreased the propagation rate of the worms and increased the time to reach full infection. But, the RL+LA scheme is limited in not stopping the propagation of the worm in the internal LAN. Hence a more sophisticated mechanism needs to be integrated into the RL+LA scheme in order to completely stop the propagation of the worm. Furthermore, this chapter has presented the discussion of the likely impact of network properties on the RL+LA countermeasure experimental results, memory overhead in case of deploying RL+LA countermeasure on the gateway devices and applicability of RL+LA Class A experimental test results on the Internet scale. The next chapter will conclude the research work reported in this thesis and will also set out some possible directions for future work.

7 CONCLUSIONS

7.1 Chapter Introduction

The research reported in this thesis sets out to answer two research questions as set out in section 2.6. This chapter sets out the conclusions of the research and also endeavours to document the extent to which original contributions may have been generated, and the original research questions have been addressed. It also provides some possible directions for future work and lists the publications generated in the course of the reported research.

7.1.1 Chapter Layout

This chapter begins with the suggested original contributions of the thesis, set out in section 7.2 based around the three research questions defined in section 2.6. Section 7.3 presents some possible directions for future work. Section 7.4 provides the list of publications generated during the research work, while section 7.5 provides the concluding statement.

7.2 Summary of Suggested Original Contributions

7.2.1 Research Question 1

Is it possible to develop and evaluate a distributed, automated worm detection, prevention and containment solution that will be more effective against fast zero-day worms than the potential solutions summarised in section 2.3? Such a countermeasure may be limited to adding delay to the worm infection time so that system administrators have additional time to patch infected hosts. It would be desirable for such a countermeasure to be able to stop the worm infection completely.

- Chapter 3 of the thesis presents the architecture and design of a distributed automated worm detection and containment scheme, termed RL+LA, that is based on the correlation of Domain Name System (DNS) queries and the destination IP address of outgoing TCP SYN and UDP datagrams leaving the network boundary. The proposed countermeasure scheme also utilizes cooperation between different communicating scheme members using a custom protocol, which was

termed as *Friends*. The absence of a DNS lookup action prior to an outgoing TCP SYN or UDP datagram to a new destination IP addresses is used as a behavioural signature for a rate limiting mechanism while the *Friends* protocol spreads reports of the event to potentially vulnerable uninfected peer networks within the scheme. A fully functional prototype was developed in the C programming language.

- Chapters 5 and 6 of the thesis present a comprehensive set of experimental work with a range of empirical experiments for evaluating the proposed countermeasure and, compares the results with those for a simple rate limiting (RL) mechanism, in the cases worm outbreaks using the PWD with attributes similar to Slammer and Witty. The results conclude that the proposed RL+LA scheme outperforms a simple RL based mechanism on small scale and on class A scale networks.

7.2.2 Research Question 2

Is it possible to develop a pseudo worm daemon with characteristics such as random and hit-list scanning, configurable rate of propagation and confinement within defined network space to allow a developed countermeasure to be empirically tested and evaluated?

- Chapter 4 of the thesis presents the architecture and design of a Pseudo-Worm Daemon (PWD) with random scanning and hit-list worm like functionality, which is implemented in the C programming language. The novelty of this worm demon is its UDP based propagation, user-configurable random scanning pool, ability to contain user defined hit-list, authentication before infecting vulnerable hosts and efficient logging of time of infection.
- Chapter 4 of the thesis also presents evaluation of PWD by conducting a series of Pseudo-Slammer and Pseudo-Witty worm experiments with real outbreak attributes of Slammer and Witty worms; and comparing Pseudo-Slammer and Pseudo-Witty worms results with real outbreak data (which is available from CAIDA). It is concluded that PWD can be used as an effective tool to empirically analyse the propagation

behaviour of random scanning and hit-list worms, and to test potential countermeasures.

7.3 Recommendations for Future Work

7.3.1 The Rate Limiting + Leap Ahead (RL+LA) Scheme

1) It will be useful to explore the effect of the designed countermeasure scheme in the presence of background traffic with a view to assess the false positive rate.

2) The containment scheme proposed by the author in chapter 3 of this thesis does not detect worm scanning activity within the cell or L2L intra-cell worm scanning activity. In order to address this limitation, ARP as a behaviour signature could be applied in the prototype for detecting worm scanning activity along with the Friends Protocol to send alerts to peer networks.

3) The current RL+LA prototype spreads worm outbreak warnings to a set of predefined friends. The Friends protocol algorithm could be enhanced to define different types of warnings with different severity levels requiring different automated responses.

4) The RL+LA prototype reported in this research uses simple username and password based authentication for the Friends protocol warnings. This could be enhanced to implement stronger security mechanisms such as IPSec (Frankel and Krishnan, 2011).

7.4 List of Publications

The following is a list of publications which have been generated to disseminate the research reported in this thesis:

7.4.1 Published Papers

[1] Shahzad K and Woodhead S, “Empirical Analysis of The Rate Limiting + Leap Ahead (RL+LA) Countermeasure against Witty Worm", in The 2015 IEEE International Symposium on Advances in Autonomic and Secure Computing and Communications (ASCC-2015), Liverpool, UK, 2015.

[2] Shahzad K and Woodhead S, “Empirical Analysis of An Improved Countermeasure against Computer Network Worms", in The IEEE Sixth International Conference on Computing, Communications and Networking Technologies (6th ICCCNT), Texas, USA, 2015.

[3] Tidy L, Shahzad K, Ahmad M, and Woodhead S, “An Assessment of the Contemporary Threat Posed by Network Worm Malware", in The Ninth International Conference on Systems and Networks Communications (ICSNC 2014), Nice, France, 12–16, October 2014.

[4] Shahzad K and Woodhead S, “A Pseudo-Worm Daemon (PWD) for Empirical Analysis of Zero-Day Network Worms and Countermeasure Testing", in The IEEE Fifth International Conference on Computing, Communications and Networking Technologies (5th ICCCNT), Hefei, Anhui, China, 2014.

[5] Shahzad K and Woodhead S, “Towards Automated Distributed Containment of Zero-Day Network Worms", in The IEEE Fifth International Conference on Computing, Communications and Networking Technologies (5th ICCCNT), Hefei, Anhui, China, 2014.

[6] Shahzad K, Woodhead S, and Bakalis P, “A Virtualized Network Testbed for Zero-Day Worm Analysis and Countermeasure Testing", in Proceedings of Advances in Security of Information and Communication Networks, Springer, Cairo, Egypt, 2013, pp. 54–64.

7.5 Chapter Summary

This chapter has presented the conclusions of the two research questions undertaken and documented the summary of suggested original contributions. It has also presented the key directions of future work and lists the publications generated in the course of the reported research.

8 BIBLIOGRAPHY

Anagnostakis, K., Greenwald, M., Ioannidis, S., Keromytis, A. and Li, D. (2003) 'A cooperative immunization system for an untrusting Internet', 11th IEEE International Conference on Networks (ICON2003), Sydney, Australia, pp. 403-408.

Antonatos, S., Akritidis, P., Markatos, E.P. and Anagnostakis, K. (2007) 'Defending against Hitlist Worms using network address space randomization', *ELSEVIER Computer Networks*, vol. 51, no. 12, August, pp. 3471-3490.

Beast 2.07 (2004), [Online], Available: https://sites.google.com/site/codenuker2k/beast207 [10th March 2015].

Benzel, T., Braden, R., Kim, D. and Neuman, C. (2007) 'Design, deployment and Use of the DETER testbed', DETER Community Workshop on Cyber Security Experimentation and Test 2007, Berkeley, CA, USA, pp. 1–8.

Briesemeister, L., Lincoln, P. and Porras, P. (2003) 'Epidemic profiles and defense of scale-free networks', The 2003 ACM Workshop on Rapid Malcode (WORM '03), Washington, DC, USA, pp. 67-75.

CAIDA: Center for Applied Internet Data Analysis (2014), [Online], Available: http://www.caida.org/ [28th March 2014].

CERT: Code Red (2001), [Online], Available: http://www.cert.org/advisories/CA-2001-19.html [16th September 2013].

CERT: Nimda Worm (2001), [Online], Available: http://www.cert.org/advisories/CA-2001-26.html [16th September 2013].

CERT: Trojan Horses (1999), [Online], Available: http://www.cert.org/historical/advisories/CA-1999-02.cfm [10th Septmber 2013].

CERT: W32/Sobig.F Worm (2003), [Online], Available: http://www.cert.org/incident_notes/IN-2003-03.html [20th September 2013].

CERT:Code Red II (2001), [Online], Available: http://www.cert.org/incident_notes/IN-2001-09.html [16th September 2013].

Chakrabarti, A. and Manimaran, G. (2002) 'Internet infrastructure security: a taxonomy', *IEEE Network*, vol. 16, no. 6, December, pp. 13–21.

Chen, Z.C., Gao, L. and Kwiat, K. (2003) 'Modeling the spread of active worms', IEEE INFOCOM 2003, San Francisco, USA, pp. 1890–1900.

Chen, Z. and Ji, C. (2005) 'A self-learning worm using importance scanning', The 2005 ACM workshop on Rapid Malcode (WORM '05), Alexandria, VA, USA, pp 22-29.

Chen, T.M. and Robert, J.-M. (2004) 'Worm epidemics in high-speed networks', *IEEE Computer*, vol. 37, no. 6, June, pp. 48-53.

Chen, S. and Tang, Y. (2004) 'Slowing down internet worms', The 24th International Conference on Distributed Computing Systems (ICDCS'04), Tokyo, Japan, pp. 312-319.

CVE - Common Vulnerabilities and Exposures (2014), [Online], Available: https://cve.mitre.org/ [14th July 2014].

CVE:CVE-2014-6271 (2014), [Online], Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271 [09th September 2014].

Daley, D.J. and Gani, J. (1999) *Epidemic modelling: an introduction*, Cambridge: Cambridge University Press.

Damn Small Linux (DSL) (2008), [Online], Available: http://www.damnsmalllinux.org [16th June 2014].

Ediger, B. (2003) *Simulating Network Worms*, [Online], Available: http://www.stratigery.com/nws/ [16th June 2013].

Ellis, D.R., Aiken, J.G., Attwood, K.S. and Tenaglia, S.D. (2004) 'A behavioral approach to worm detection', The 2004 ACM Workshop on Rapid Malcode (WORM '04), Fairfax, VA, USA, pp. 43-53.

Eset: Win32/Gnuman (2008), [Online], Available: http://www.eset.com/us/threat-center/encyclopedia/threats/gnuman/ [15th July 2015].

Fagen, W., Cangussu, J. and Dantu, R. (2009) 'A virtual environment for network testing', *ELSEVIER Journal of Network and Computer Applications*, vol. 32, no. 1, January, pp. 184-214.

Falliere, N. and Murchu, .L.O. (2011) *W32.Stuxnet Dossier*, [Online], Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf [16th September 2013].

Frankel, S. and Krishnan, S. (2011) *IP Security (IPsec) and Internet Key Exchange (IKE) document roadmap*, [Online], Available: https://tools.ietf.org/html/rfc6071 [24th September 2014].

Frauenthal, J.C. (1980) *Mathematical modeling in epidemiology*, Springer-Verlag.

8. BIBLIOGRAPHY

F-Secure: Bomber (1992), [Online], Available: https://www.f-secure.com/v-descs/bomber.shtml [23rd August 2015].

F-Secure: Net-Worm:W32/Santy.A (2004), [Online], Available: http://www.f-secure.com/v-descs/santy_a.shtml [17th September 2013].

Gaespy Archade (1999), [Online], Available: http://www.gamespyarcade.com/ [10th September 2013].

Ganger, G., Economou, G. and Bielski, S. (2002) *Self-securing network interfaces: what, why, and how*, Pittsburgh, Pennsylvania, United States: Carnegie Mellon University.

GNU Zebra (2005), [Online], Available: https://www.gnu.org/software/zebra/ [07th March 2015].

Gorman, S., Kulkarni, R., Schintler, L. and Stoug, R. (2003) *Least effort strategies for cybersecurity*, George Mason University.

Holz, T., Steiner, M., Dahl, F., Biersack, E. and Freiling, F. (2008) 'Measurements and mitigation of peer-to-peer-based botnets: a case study on Storm worm', First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET '08), San Francisco, CA, USA.

Jiang, X., Xu, D. and Eigenmann, R. (2004) 'Protection mechanisms for application service hosting platforms', 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), Chicago, Illinois, USA, pp. 633-639.

Jiang, X., Xu, D., Wang, H.J. and Spaffrod, E.H. (2006) 'Virtual playgrounds for worm behavior investigation', 8th International Symposium on Recent Advances in Intrusion Detection (RAID), Seattle, USA, pp. 1-21.

Jung, J., Sit, E., Balakrishnan, H. and Morris, R. (2002) 'DNS performance and the effectiveness of caching', *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, October, pp. 589-603.

Kaur, R. and Singh, M. (2014) 'Efficient hybrid technique for detecting zero-day polymorphic worms', Advance Computing Conference (IACC), Gurgaon, India, pp. 95-100.

Kermack, W.O. and McKendrick, A.G. (1927) 'A contribution to mathematical theory of epimedics', *Royal Society of London*, vol. 115, no. 772, August, pp. 700-721.

Kim, H.-A. and Karp, B. (2004) 'Autograph: toward automated, distributed worm signature detection', The 13th conference on USENIX Security Symposium (SSYM'04), San Diego, CA, USA, pp. 19-34.

8. BIBLIOGRAPHY

KrebsonSecurity (2012), [Online], Available: http://krebsonsecurity.com/2012/10/service-sells-access-to-fortune-500-firms/ [14th July 2014].

Kruegel, C., Kirda, E., Mutz, D., Robertson, W. and Vigna, G. (2005) 'Polymorphic worm detection using structural information of executables', 8th international conference on Recent Advances in Intrusion Detection (RAID'05), Seattle, WvA, USA, pp. 207-226.

Liljenstam, M., Nicol, D.M., Berk, V.H. and Gray, R.S. (2003) 'Simulating realistic network worm traffic for worm warning system design and testing ', ACM Workshop on Rapid Malcode 2003, Washington, DC, USA, pp. 24–33.

LINIXBBQ (2012), [Online], Available: http://linuxbbq.org/ [03rd March 2015].

Lippmann, R.P., Fried, D.J., Graf, I., Haines, J.W., Kendall, K.R., McClung, D., Weber, D., Webster, S.E., Wyschogrod, D., Cunningham, R.K. and Zissman, M.A. (2000) 'Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation', 2000 DARPA Information Survivability Conference and Exposition (DISCEX), New York, USA, pp. 12-26.

Li, P., Salour, M. and Su, X. (2008) 'A survey Of Internet worm detection and containment', *IEEE Communication Surverys and Tutorials*, vol. 10, no. 1, April, pp. 20-35.

Lotka, A.J. (1925) *Elements of physical biology*, Williams and Wilkins Company.

Marsaglia, G. (2003) 'Random number generators', *Journal of Modern Applied Statistical Methods*, vol. 2, no. 1, pp. 2-13.

Marsaglia, G. and Zaman, A. (1991) 'A new class of random number generators', *Annals of Applied Probability*, vol. 1, no. 3, August, pp. 462-480.

McAfee (2006), [Online], Available: http://web.archive.org/web/20060823090948/http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_akapoor_rootkits1_en.pdf [15th September 2013].

McAffe: W95/firkin.worm (2000), [Online], Available: http://home.mcafee.com/virusinfo/virusprofile.aspx?key=98557 [16th September 2013].

Mockapetris, P. (1987) *Domain Name - Concept and Facilities, IETF*, [Online], Available: http://tools.ietf.org/html/rfc1034 [04th September 2013].

8. BIBLIOGRAPHY

Mohammed, M.M.Z.E. (2012) *Automated signature generation for zero-day polymorphic worms using a double-honeynet*, PhD Thesis edition, Cape Town: University of Cape Town.

Mohammed, M.M.Z.E., Chan, H.A., Ventura, N. and Pathan, A.-S.K. (2013) 'An automated signature generation method for zero-day polymorphic worms based on multilayer perceptron model', International Conference on Advanced Computer Science Applications and Technologies, Kuching, Malaysia, pp. 450-455.

Moore, D. (2002) 'Network telescopes: observing small or distant security events', USENIX Security '02, San Francisco, USA.

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S. and Weaver, N. (2003) 'Inside the Slammer worm', *IEEE Security and Privacy*, vol. 1 , no. 4, August, pp. pp. 33-39.

Moore, D., Shannon, C., Voelker, G. and Savage, S. (2003) 'Internet quarantine: requirements for containing self-propagating code', Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies (INFOCOM 2003), San Francisco, USA, pp. 1901-1910.

Nazario, J., Ptacek, T. and Song, D. (2004) *Wormability: A description for vulnerabilities*, [Online], Available: http://repo.hackerzvoice.net/depot_madchat/vxdevl/papers/avers/wormability _researchOct04.pdf [15th October 2014].

Net Market Share: Desktop operating system market share (2014), [Online], Available: http://www.netmarketshare.com/ [20 September 2014].

Nicol, D.M. and Liljenstam, M. (2005) 'Models and analysis of active worm defense', The Third international conference on Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS'05), St. Petersburg, Russia, pp. 38-53.

Nojiri, D., Rowe, J. and Levitt, K. (2003) 'Cooperative response strategies for large scale attack mitigation', The 3rd DARPA Information Survivability Conference and Exposition, Washington, DC, USA, pp. 293-302.

Parsons, J.J. and Oja, D. (2010) *New perspectives on computer concepts 2011; Introductory*, Boston, MA, USA: Course Technology Press.

Paul, S. and Mishra, B.K. (2013) 'PolyS: Network-based signature generation for zero-day polymorphic worms', *International Journal of Grid and Distributed Computing*, vol. 6, no. 4, August, pp. 63-74.

Paxson, V. (1998) 'Bro: A system for detecting network intruders in real-time', 7th USENIX Security Symposium, Texas, USA, pp. 31–52.

Pearson, K. (1895) 'Notes on regression and inheritance in the Case of two parents', *Proceeding of the Royal Society of London*, vol. 58, no. 1, January, pp. 240-242.

Perumalla, K.S. and Sundaragopalan, S. (2004) 'High fidelity modeling of computer network worms', 20th Annual Computer Security Applications Conference (ACSAC), Tucson, AZ, USA, pp. 126–135.

PJSIP: PJLIB Library (2008), [Online], Available: http://www.pjsip.org/pjlib/docs/html/ [13th September 2013].

Porras, P., Briesemeister, L., Skinner, K., Levitt, K., Rowe, J. and Ting, Y.-C.A. (2004) 'A hybrid quarantine defense', The 2004 ACM workshop on Rapid malcode (WORM '04), Fairfax, VA , USA, pp. 73-82.

Provos, N. (2004) 'A virtual honeypot framework', USENIX 13th security symposium, San Diego, USA, pp. 1-14.

Puppy Linux (2003), [Online], Available: http://www.puppylinux.com/ [03rd March 2015].

Quagga Routing Suite (1999), [Online], Available: http://www.nongnu.org/quagga [16th June 2013].

Ramneek, P. (2003) *Bots &; Botnet: An overview*, [Online], Available: http://www.sans.org/reading-room/whitepapers/malicious/bots-botnet-overview-1299 [03rd August 2003].

Richmond, M. (2006) 'Digital forensic reconstruction and the virtual security testbed ViSe', Conference on Detection of Intrusions and Malware, Paris, France, pp. 144-163.

Riordan, J., Zamboni, D. and Duponchel, Y. (2006) 'Building and deploying Billy Goat, a worm-detection system', The 18th Annual FIRST Conference, Baltimore, USA, pp. 1-12.

Rossey, L.M., Cunningham, R.K., Fried, D.J., Rabek, J.C. and Lippmann, R.P. (2002) 'LARIAT: Lincoln adaptable real time information assurance testbed', IEEE Aerospace Conference, Big Sky, Montana, USA, pp. 2671-2682.

Schechter, S.E., Jung, J. and Berger, A.W. (2004) 'Fast detection of scanning worm infections', 7th International Symposium on Recent Advances in Intrusion Detection, (RAID 2004), Sophia Antipolois, France, pp. 59-81.

Shahzad, K. and Woodhead, S. (2014a) 'Towards automated distributed containment of zero-day network worms', The IEEE Fifth International

Conference on Computing, Communications and Networking Technologies (5th ICCCNT), Hefei, Anhui, China.

Shahzad, K. and Woodhead, S. (2014b) 'A Pseudo-Worm Daemon (PWD) for empirical analysis of zero-day network worms and countermeasure testing', The IEEE Fifth International Conference on Computing, Communications and Networking Technologies (5th ICCCNT), Anhui, China.

Shahzad, K., Woodhead, S. and Bakalis, P. (2013) 'A virtualized network testbed for zero-day worm analysis and countermeasure testing', The First International Coneferene on Advances in Security of Information and Communication Networks (SecNet2013), Cario, Egypt, pp. 54-64.

Shannon, C. and Moore, D. (2004) 'The spread of the Witty worm', *IEEE Security and Privacy*, vol. 2, no. 4, October, pp. 46-50.

SHODAN - Computer Search Engine (2009), [Online], Available: http://www.shodanhq.com/ [10th July 2014].

Singh, S., Estan, C., Varghese, G. and Savage, S. (2004) 'Automated worm fingerprinting', The 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04), San Franscisco, CA, USA, pp. 4-19.

Snort (1998), [Online], Available: http://www.snort.org/ [16th June 2013].

Software Insider (2013), [Online], Available: http://virtualization.softwareinsider.com/ [16th June 2013].

Staniford, S., Moore, D., Paxson, V. and Weaver, N. (2004) 'The top speed of flash worms', The 2004 ACM workshop on Rapid Malcode (WORM '04), Fairfax, VA, USA, pp. 33-42.

Staniford, S., Paxson, V. and Weaver, N. (2002) 'How to own the Internet in your spare time', The 11th USENIX Security Symposium, San francisco, CA, USA, pp. 149-167.

Sun, W., Katta, V., Krishna, K. and Sekar, R. (2008) 'V-Netlab: An approach for realizing logically isolated networks for security experiments', Conference on Cyber Security Experimentation and Test (CSET ’08), Berkeley, CA, USA, pp. 1-6.

Symantec: JS.Gigger.A@mm (2002), [Online], Available: http://www.symantec.com/security_response/writeup.jsp?docid=2002-011011-3021-99 [17th September 2013].

Symantec: W32.Welchia.Worm (2003), [Online], Available: http://www.symantec.com/security_response/writeup.jsp?docid=2003-081815-2308-99 [10th March 2014].

8. BIBLIOGRAPHY

Symantec: W95.CIH (1998), [Online], Available: http://www.symantec.com/security_response/writeup.jsp?docid=2000-122010-2655-99 [23rd August 2015].

TCPDUMP & LIBPCAP (2008), [Online], Available: http://www.tcpdump.org [16th June 2013].

The BIRD Internet Routing Daemon (2008), [Online], Available: http://bird.network.cz/ [03rd March 2015].

The netfilter.org "iptables" project (1998), [Online], Available: http://www.netfilter.org/projects/iptables/index.html [16th June 2013].

Tidy, L., Shahzad, K., Ahmad, M.A. and Woodhead, S. (2014) 'An assessment of the contemporary threat posed by network worm malware', The Ninth International Conference on Systems and Networks Communications (ICSNC 2014), Nice, France, 12-16, pp. 92-98.

Tidy, L., Woodhead, S. and Wetherall, J. (2013) 'A large-scale zero-day worm simulator for cyber-epidemiological analysis', *International Journal of Advances in Computer Networks and Security*, vol. 3, pp. 69-73.

TIME Magazine (2002), [Online], Available: http://content.time.com/time/specials/packages/article/0,28804,1991915_1991909_1991762,00.html [10 March 2015].

Tiny Core Linux (2009), [Online], Available: http://tinycorelinux.net/ [03rd March 2015].

Toyoizumi, H. and Kara, A. (2002) 'Predators: good mobile code combat against computer viruses', The 2002 Workshop on New security Paradigms (NSPW '02), Virginia Beach, Virginia, USA, pp. 11 -17.

ubuntu (2004), [Online], Available: http://www.ubuntu.com/ [10th September 2014].

Vahdat, A., Yocum, Y., Walsh, K. and Mahadev, P. (2002) 'Scalability and accuracy in a large-scale network emulator', USENIX 5th symposium on Operating Systems Design and Implementation (OSDI), Boston, MA,USA, pp. 271–284.

VMware (1998), [Online], Available: http://www.vmware.com/uk/products/ [25th September 2014].

VMware Academic Program (VMAP) (2010), [Online], Available: http://www.vmware.com/partners/academic/program-overview.html [10th September 2014].

8. BIBLIOGRAPHY

VMware ESXi (2010), [Online], Available: http://www.vmware.com/products/vsphere/esxi-and-esx/index.html [16th September 2013].

VMware Security Advisories (2013), [Online], Available: https://www.vmware.com/security/advisories/VMSA-2013-0006.html [14th July 2014].

VMware vCenter Server (2012), [Online], Available: http://www.vmware.com/products/vcenter-server/overview.html [16th September 2013].

VMware vSphere PowerCLI 5.0 (2011), [Online], Available: http://communities.vmware.com/community/vmtn/server/vsphere/automationtools/powercli?view=overview [16th September 2013].

Volterra, V. (1926) *Chapman R. N. 1931. Animal Ecology*, pp. 409-448.

W3Counter (2014), [Online], Available: http://www.w3counter.com/globalstats.php [14th July 2014].

Wang, K., Cretu, G. and Stolfo, S.J. (2005) 'Anomalous payload-based worm detection and signature generation', The 8th international conference on Recent Advances in Intrusion Detection (RAID'05), Seattle, WA, USA, pp. 227-246.

Wang, L., Li, Z., Chen, Y., Fu, Z.(. and Li, X. (2010) 'Thwarting zero-day polymorphic worms with network-level length-based signature generation', *IEEE/ACM Transactions on Networking*, vol. 18, no. 1, August , pp. 53-66.

Weaver, N., Paxson, V., Staniford, S. and Cunningham, R. (2003) 'A taxonomy of computer worms', The 2003 ACM workshop on Rapid Malcode (WORM '03), New York, NY, USA, pp. 11-18.

Weaver, N., Staniford, S. and Paxson, V. (2004) 'Very fast containment of scanning worms', 13th USENIX Security Symposium, San Diego, CA, USA, pp. 29–44.

White, B., Lepreau, J., Stoller, L., Ricci, R. and Gruprasad, S. (2002) 'An integrated experimental environment for distributed systems and networks', 5th Symposium on Operating Systems Design and Implementation, Boston, MA, USA, pp. 265–270.

Whyte, D., Kranakis, E. and Oorschot, P.C.v. (2005) 'DNS-based detection of scanning worms in an enterprise network', 12th Annual Network and Distributed Systems Symposium (NDSS), San Diego, California, USA.

Whyte, D., van Oorschot, P.C. and Kranakis, E. (2005) 'Detecting intra-enterprise scanning worms based on address resolution', ACSAC '05

Proceedings of the 21st Annual Computer Security Applications Conference, Tucson, Arizona, USA, pp. 371-380.

Williamson, M.M. (2002) 'Throttling viruses: restricting propagation to defeat malicious mobile code', The 18th Annual Computer Security Applications (ACSAC '02), Las Vegas, NV, USA, pp. 61-68.

Wong, C., Bielski, S., Studer, A. and Wang, C. (2005) 'Empirical analysis of rate limiting mechanisms', 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), Seattle, WA, USA, pp. 22-42.

Xiang, Y., Fan, X. and Zhu, W.T. (2009) 'Propagation of active worms: a survey', *International Journal of Computer Systems Science and Engineering*, vol. 24, no. 3, May, pp. 157-172.

Zamboni, D., Riordan, J. and Yates, M. (2007) 'Boundary detection and containment of a local worm infections', The 3rd USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI'07), Santa Clara, CA.

Ziyad, S.A.-S. (2011) *Topology-aware vunerablity mitigation worms*, PhD Thesis edition, London: Royal Holloway, University of London.

Zou, C.C., Gao, L., Gong, W. and Towsley, D. (2003) 'Monitoring and early warning for internet worms', The 10th ACM Conference on Computer and Communications Security (CCS '03), Washington, DC, USA, pp. 190-199.

Zou, C.C., Gong, W. and Towsley, D. (2002) 'Code Red worm propagation modeling and analysis', 9th ACM Conference on Computer and Communication Security, Washington, DC, USA, pp. 138-147.

Zou, C.C., Towsley, D., Gong, W. and Cai, S. (2005) 'Routing worm: A fast, selective attack worm based on IP address information', Workshop on Principles of Advanced and Distributed Simulation (PADS 2005), Monterey, CA, USA, pp 199-206.

9 APPENDICES

9.1 The RL+LA Source Code

db.c

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "db.h"
#include "dns_sniff.h"

List dns_table;
List tcp_udp_table;

unsigned int thresholdValue;

void dbInit(void)
{
  initList(&dns_table);
  initList(&tcp_udp_table);

  thresholdValue = THRCNT;
}

void insertNewDNSReq(struct in_addr ip_src)
{
  addEndUnique(&dns_table, ip_src.s_addr);
}

void insertNewTCP_UDPEntry(struct in_addr ip_src)
{
    char iptables_block_ip_cmd_in[128];
    char iptables_block_ip_cmd_out[128];

    struct in_addr dst_router;

    if(!containsElem(&dns_table, ip_src.s_addr))
    {
      addEndUnique(&tcp_udp_table, ip_src.s_addr);
      display(&tcp_udp_table);

      if(incrementCntElemt(&tcp_udp_table, ip_src.s_addr,
thresholdValue))
      {
        printf("Host: %s must be blocked \n",
inet_ntoa(ip_src));

        //snprintf(iptables_block_ip_cmd_in, 127, "iptables -I
INPUT -s %s -j DROP", inet_ntoa(ip_src));
        snprintf(iptables_block_ip_cmd_out, 127, "iptables -I
OUTPUT -s %s -j DROP", inet_ntoa(ip_src));

            #ifdef HALF_THR
            thresholdValue /= 2;
            #endif
```

```
        if( system(iptables_block_ip_cmd_in) != 0 )
        {
              perror("cannot execute iptable block command !");
        }

        if( system(iptables_block_ip_cmd_out) != 0 )
        {
              perror("cannot execute iptable block command !");
        }

        #ifdef SEND_FRIEND_MSG
        if( (strcmp(if_address_str, IP_ROUTER_A)!=0) )
        {
              inet_aton(IP_ROUTER_A, &dst_router);
              sendTheRequestToPeer(dst_router, ip_src);
        }

        if( (strcmp(if_address_str, IP_ROUTER_B)!=0) )
        {
              inet_aton(IP_ROUTER_B, &dst_router);
              sendTheRequestToPeer(dst_router, ip_src);
        }

        if( (strcmp(if_address_str, IP_ROUTER_C)!=0) )
        {
              inet_aton(IP_ROUTER_C, &dst_router);
              sendTheRequestToPeer(dst_router, ip_src);
        }
        if( (strcmp(if_address_str, IP_ROUTER_E)!=0) )
        {
              inet_aton(IP_ROUTER_E, &dst_router);
              sendTheRequestToPeer(dst_router, ip_src);
        }

        if( (strcmp(if_address_str, IP_ROUTER_F)!=0) )
        {
              inet_aton(IP_ROUTER_F, &dst_router);
              sendTheRequestToPeer(dst_router, ip_src);
        }

        if( (strcmp(if_address_str, IP_ROUTER_G)!=0) )
        {
              inet_aton(IP_ROUTER_G, &dst_router);
              sendTheRequestToPeer(dst_router, ip_src);
        }
        #endif

      }
    }
}

void checkEntriesInTcpUdpTable(unsigned char cmd)
{
      #ifdef DUMP_MEM
      printf("DNS table: \n");
      display(&dns_table);
      printf("\n");

      printf("TCP_UDP table: \n");
      display(&tcp_udp_table);
      printf("\n");
```

```
	#endif

	if(cmd == UPDATE_CNT)
	{
	  updateCntValue(&tcp_udp_table);
	}
}
```

db.h

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/


#ifndef DB_H
#define DB_H

#include <netinet/in.h>
#include "list.h"

#define UPDATE_CNT 0
#define BLOCK_ENTRIES 1

#define CNT_MAX 1000

void dbInit(void);
void insertNewDNSReq(struct  in_addr ip_src);
void insertNewTCP_UDPEntry(struct in_addr ip_src);
void checkEntriesInTcpUdpTable(unsigned char cmd);

#endif
```

dns_sniff.c

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/


#include "dns_sniff.h"


#ifdef IS_BORDER_ROUTER
char* external_interface;
#endif

int main(int argc, char* argv[])
{
#ifndef IS_BORDER_ROUTER

	if(argc < 2)
	{
		printf("\nUsage: ./dns_sniff.o eth_interface
<debug>\n\n");
		return(EXIT_FAILURE);
	}
#else
	if(argc < 3)
	{
		printf("\nUsage: ./dns_sniff.o
eth_internal_interface eth_external_interface <debug>\n\n");
		return(EXIT_FAILURE);
	}
```

```
	external_interface=argv[2];
#endif

	initProgram();
	open_device(argv[1]);

	return 0;
}
```

dns_sniff.h

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/

#ifndef DNS_SNIFF
#define DNS_SNIFF
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pjlib.h>
#include <pjlib-util.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <linux/if.h>
#include "db.h"

struct white_list_elem
{
#define NUMBER_OF_WHITE_LIST_ELEMS 1
	char white_ip_src[20];
	char white_ip_dst[20];
	int white_sport;
	int white_dport;
};

#define DUMP_MEM
#define SEND__MSG
#define HALF_THR

#define IS_NO_BORDER_ROUTER
#define DEBUG
//#define NO_DEBUG
#define NSEC 30
#define NSEC_BLOCK 15
#define THRCNT 5
#define SIGTIMER	(SIGRTMAX)
#define MAX_BYTES_TO_CAPTURE 2048
#define DNS_TABLE_SIZE 1024
#define SIZE_ETHERNET 14
#define DNS_HEADER_SIZE 12
#define ETHER_ADDR_LEN	6
#define ENCR_KEY 0xBAF4
#define ROUTER_USERNAME "router"
#define ROUTER_PASSWORD "passwd"
```

```
#define ROUTER_COMMAND  "half the threashold"
#define IP_ROUTER_A "10.0.0.1"
#define IP_ROUTER_B "10.64.0.1"
#define IP_ROUTER_C "10.128.0.1"
#define IP_ROUTER_E "10.192.0.1"
//#define IP_ROUTER_F "192.168.5.1"
//#define IP_ROUTER_G "192.168.6.1"

extern char* if_address_str;
extern unsigned int thresholdValue;

#ifdef IS_BORDER_ROUTER
      extern int number_of_friends;
      extern char* list_of_friends[];
      extern char* external_interface;
#endif

/* Ethernet header */
struct sniff_ethernet {
        u_char  ether_dhost[ETHER_ADDR_LEN];
        u_char  ether_shost[ETHER_ADDR_LEN];
        u_short ether_type;
};

/* IP header */
struct sniff_ip {
        u_char  ip_vhl;
        u_char  ip_tos;
        u_short ip_len;
        u_short ip_id;
        u_short ip_off;
        #define IP_RF 0x8000
        #define IP_DF 0x4000
        #define IP_MF 0x2000
        #define IP_OFFMASK 0x1fff
        u_char  ip_ttl;
        u_char  ip_p;
        u_short ip_sum;
        struct  in_addr ip_src,ip_dst;
};
#define IP_HL(ip)                (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                 (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp {
        u_short th_sport;
        u_short th_dport;
        tcp_seq th_seq;
        tcp_seq th_ack;
        u_char  th_offx2;
      #define TH_OFF(th)    (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
        #define TH_RST  0x04
        #define TH_PUSH 0x08
        #define TH_ACK  0x10
        #define TH_URG  0x20
        #define TH_ECE  0x40
```

```
        #define TH_CWR  0x80
        #define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
        u_short th_win;
        u_short th_sum;
        u_short th_urp;
};

/* UDP header */
struct sniff_udp {
        u_short th_sport;
        u_short th_dport;
      u_short th_length;
      u_short th_sum;
};
int open_device(char* dev);
int initProgram(void);
void processPacket(u_char* arg, const struct pcap_pkthdr*
pkthdr, const u_char* packet);
void printPayload(const u_char* packet, int indx, int length);

timer_t SetTimer(int signo, int sec, int mode);
void SignalHandler(int signo, siginfo_t * info, void *context);
void getLocalMACAddress(char* dev);
void getInterfaceIPAddress(char* dev);
void sendTheRequestToPeer(struct in_addr dst_router, struct
in_addr ip_to_block);
void encrypt(char* str,int key);
void decrypt(char* str,int key);
int isCommandFromFriends(char* ip_src_str);
void executeRemoteCommand(const u_char* packet, int
payload_begin, char* ip_src_str);
int isInTheWhiteList(char* ip_src_str, char* ip_dst_str, int
sport, int dport);

#endif
```

dns_sniff_func.c

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/


#include "dns_sniff.h"

#include <sys/ioctl.h>

pcap_t *handle=NULL;

pj_pool_factory *mem;
pj_pool_t *pool;
pj_caching_pool caching_pool;

timer_t timerid;
int timer_cnt;

struct in_addr if_address;
unsigned char if_mac_str[13];
char* if_address_str;

struct white_list_elem whiteList[NUMBER_OF_WHITE_LIST_ELEMS] =
{{"64.4.9.254", "192.168.127.129", 80, -1},
```

```
                              };

#ifdef IS_BORDER_ROUTER
      int number_of_friends=4;
      char*
list_of_friends[]={"10.0.0.1","10.64.0.1","10.128.0.1","10.192.
0.1"};
#endif

int initProgram(void)
{
      int i=0;
      timer_cnt = 0;
      struct sigaction sigact;

      dbInit();

      mem = &caching_pool.factory;
      pj_init();

        sigemptyset(&sigact.sa_mask);
        sigact.sa_flags = SA_SIGINFO;
        sigact.sa_sigaction = SignalHandler;

        if (sigaction(SIGTIMER, &sigact, NULL) == -1) {
                perror("sigaction failed");
                return -1;
        }

        sigaction(SIGINT, &sigact, NULL);

        timerid = SetTimer(SIGTIMER, NSEC*1000, 1);
}

int open_device(char* dev)
{
      int count=0;
      char errbuf[PCAP_ERRBUF_SIZE];
      bpf_u_int32 mask;
      bpf_u_int32 net;
      char filter_exp[]="ip";
      struct bpf_program dns_filter;
      struct in_addr router_ip;

      handle = pcap_open_live(dev, MAX_BYTES_TO_CAPTURE, 1,
512, errbuf);
      if (handle == NULL) {
            fprintf(stderr, "Couldn't open the device: %s\n",
errbuf);
            return(EXIT_FAILURE);
      }


      if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
            fprintf(stderr, "Couldn't get netmask for device
%s: %s\n", dev, errbuf);
            net = 0;
            mask = 0;
      }

      if (pcap_compile(handle, &dns_filter, filter_exp, 0,
```

```
mask) == -1) {
            fprintf(stderr, "ERROR: %s\n",
pcap_geterr(handle));
            return(EXIT_FAILURE);
      }

            if (pcap_setfilter(handle, &dns_filter) == -1) {
            fprintf(stderr, "ERROR: %s\n",
pcap_geterr(handle));
            return(EXIT_FAILURE);
      }

      getLocalMACAddress(dev);
      getInterfaceIPAddress(dev);

      if_address_str=strdup(inet_ntoa(if_address));

      printf("Sniffing interface %s (%s) ...\n", dev,
if_address_str);

      if ( pcap_loop(handle, -1, processPacket,
(u_char*)&count) < 0 )
      {
            fprintf(stderr, "ERROR: %s\n",
pcap_geterr(handle));
            return(EXIT_FAILURE);
      }

      /* cleanup */
      pcap_freecode(&dns_filter);
      pcap_close(handle);
}

void processPacket(u_char* arg, const struct pcap_pkthdr*
pkthdr, const u_char* packet)
{
      static int count = 1;
      const struct sniff_ethernet *ethernet;
      const struct sniff_ip *ip;
      const struct sniff_tcp *tcp;
      const struct sniff_udp *udp;
      int size_payload;
      int size_ip;
      int size_tcp;
      int size_udp=8;
      int dport, sport;
      int domain_name_pos;
      int type_pos;
      unsigned short dns_type, ansrr_cnt, addrr_cnt;
      int result, i;

      char* dname;
      char* ip_dns_reply;
      char* ip_src_str;
      char* ip_dst_str;
      pj_status_t status;
      pj_dns_parsed_packet *dns_pkt;

      pj_caching_pool_init( &caching_pool,
&pj_pool_factory_default_policy, 0 );
      pool = pj_pool_create(mem, NULL, 2000, 2000, NULL);
```

```
	#ifdef NO_DEBUG
	printf("\nPacket number %d:\n", count);
	#endif
	count++;

	/* define ethernet header */
	ethernet = (struct sniff_ethernet*)(packet);

	/* define/compute ip header offset */
	ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
	size_ip = IP_HL(ip)*4;
	if (size_ip < 20) {
		#ifdef NO_DEBUG
		printf("   * Invalid IP header length: %u bytes\n",
size_ip);
		#endif
		return;
	}

	ip_src_str=strdup(inet_ntoa(ip->ip_src));
	ip_dst_str=strdup(inet_ntoa(ip->ip_dst));

	#ifdef NO_DEBUG
	printf("       From: %s\n", inet_ntoa(ip->ip_src));
	printf("         To: %s\n", inet_ntoa(ip->ip_dst));
	#endif

	/* determine protocol */
	switch(ip->ip_p) {
		case IPPROTO_TCP:
			tcp = (struct sniff_tcp*)(packet +
SIZE_ETHERNET + size_ip);

			sport=ntohs(tcp->th_sport);
			dport=ntohs(tcp->th_dport);

			size_tcp = TH_OFF(tcp)*4;
			#ifdef NO_DEBUG
			printf("   Protocol: TCP\n");
			printf("       From: %s on %d\n", ip_src_str,
sport);
			printf("         To: %s on %d\n", ip_dst_str,
dport);
			#endif

			#ifndef IS_BORDER_ROUTER

			if(( isInTheWhiteList(ip_src_str, ip_dst_str,
sport, dport) == 1 ) || (strcmp(if_address_str, ip_src_str) ==
0))
			{
				return;
			}
			#endif

			break;
		case IPPROTO_UDP:
			udp = (struct sniff_udp*)(packet +
SIZE_ETHERNET + size_ip);
```

```
                  sport=ntohs(udp->th_sport);
                  dport=ntohs(udp->th_dport);

                  size_payload = ntohs(ip->ip_len) - (size_ip +
size_udp);

                  #ifdef NO_DEBUG
                  printf("   Protocol: UDP\n");
                  printf("       From: %s on %d\n", ip_src_str,
sport);
                  printf("         To: %s on %d\n", ip_dst_str,
dport);
                  #endif

                  #ifndef IS_BORDER_ROUTER
                                  if((
isInTheWhiteList(ip_src_str, ip_dst_str, sport, dport) == 1 )
|| (strcmp(if_address_str, ip_src_str) == 0))
                  {
                        return;
                  }
                  #endif

                  break;
            case IPPROTO_ICMP:
                  return;
            case IPPROTO_IP:
                  #ifdef NO_DEBUG
                  printf("   Protocol: IP\n");
                  #endif
                  break;
            default:
                  #ifdef NO_DEBUG
                  printf("   Protocol: unknown\n");
                  #endif
                  return;
      }

#ifndef IS_BORDER_ROUTER

      if ( (dport == 53) && (ip->ip_p == IPPROTO_UDP) )
      {
            return;
      }

      if ( (sport == 53) && (ip->ip_p == IPPROTO_UDP) )
      {

      status = -1;
            dns_pkt = NULL;

            status = pj_dns_parse_packet(pool,
(packet+SIZE_ETHERNET + size_ip +
size_udp),(unsigned)(size_payload), &dns_pkt); // plase
eloborate

            if( status == PJ_SUCCESS )
            {
                  /* get the number of answers */
                  ansrr_cnt=dns_pkt->hdr.anscount;
```

```
                  addrr_cnt=dns_pkt->hdr.arcount;

                  /* look into the answers list and check for
the ip */

                  for (i=0; i<ansrr_cnt; i++)
                  {
                        //printf("%s -> ",dns_pkt-
>ans[i].name.ptr);

                        if ( dns_pkt->ans[i].type ==
PJ_DNS_TYPE_A )
                        {
                              //printf("%s\n",
pj_inet_ntoa(dns_pkt->ans[i].rdata.a.ip_addr));

      ip_dns_reply=strdup(pj_inet_ntoa(dns_pkt-
>ans[i].rdata.a.ip_addr));
      insertNewDNSReq(ip->ip_dst);
                        }
                  }

                  for (i=0; i<addrr_cnt; i++)
                  {
                        //printf("%s -> ",dns_pkt-
>arr[i].name.ptr);

                        if ( dns_pkt->arr[i].type ==
PJ_DNS_TYPE_A )
                        {
                              //printf("%s\n",
pj_inet_ntoa(dns_pkt->arr[i].rdata.a.ip_addr));

      ip_dns_reply=strdup(pj_inet_ntoa(dns_pkt-
>arr[i].rdata.a.ip_addr));
                              //insertNewDNSReq(ip_dst_str,
dns_pkt->arr[i].name.ptr, ip_dns_reply);
                              insertNewDNSReq(ip->ip_dst);
                        }
                  }

                  //pj_dns_dump_packet(dns_pkt);
            }

         }
      else if ( ip->ip_p == IPPROTO_UDP ) // some UDP package
      {


            if( strcmp(if_address_str, ip_dst_str ) == 0 &&
isCommandFromFriends(ip_src_str) )
            {

                  //printf("       From: %s\n", ip_src_str);
                  //printf("         To: %s\n", ip_dst_str);

                  executeRemoteCommand(packet, (SIZE_ETHERNET +
size_ip + size_udp), ip_src_str);
            }
            else if (pkthdr->direction == PCAP_D_OUT)
            {
```

```
            insertNewTCP_UDPEntry(ip->ip_src);
            }
      }
      else if ( ip->ip_p == IPPROTO_TCP ) // TCP package
      {
            if (size_tcp < 20) {
                  #ifdef NO_DEBUG
                  printf("   * Invalid TCP header length: %u
bytes\n", size_tcp);
                  #endif
            }
            else
            {

                  if(pkthdr->direction == PCAP_D_OUT)
                  {
                  insertNewTCP_UDPEntry(ip->ip_src);
                  }
            }
      }

#else
      if ( ip->ip_p == IPPROTO_UDP )
      {
            if( isCommandFromFriends(ip_src_str) )
            {
                  executeRemoteCommand(packet, (SIZE_ETHERNET +
size_ip + size_udp), ip_src_str);
            }
      }
#endif


      return;
}


void printPayload(const u_char* packet, int indx, int length)
{
      int i;

      for( i=indx; i<length; i++ )
      {
            if( isprint(packet[i]) )
            {
                  printf("%c ",packet[i]);
            }
            else
            {
                  printf(".");
            }
      }
}

timer_t SetTimer(int signo, int sec, int mode)
{
        struct sigevent sigev;
        timer_t timerid;
        struct itimerspec itval;
        struct itimerspec oitval;
```

```
        sigev.sigev_notify = SIGEV_SIGNAL;
        sigev.sigev_signo = signo;
        sigev.sigev_value.sival_ptr = &timerid;

        if (timer_create(CLOCK_REALTIME, &sigev, &timerid) ==
0) {
                itval.it_value.tv_sec = sec / 1000;
                itval.it_value.tv_nsec = (long)(sec % 1000) *
(1000000L);

                if (mode == 1) {
                        itval.it_interval.tv_sec =
itval.it_value.tv_sec;
                        itval.it_interval.tv_nsec =
itval.it_value.tv_nsec;
                } else {
                        itval.it_interval.tv_sec = 0;
                        itval.it_interval.tv_nsec = 0;
                }

                if (timer_settime(timerid, 0, &itval, &oitval)
!= 0) {
                        perror("time_settime error!");
                }
        } else {
                perror("timer_create error!");
                return -1;
        }
        return timerid;
}


void SignalHandler(int signo, siginfo_t * info, void *context)
{
        if (signo == SIGTIMER) {
            if(timer_cnt == NSEC_BLOCK)
            {
                  checkEntriesInTcpUdpTable(BLOCK_ENTRIES);
                  timer_cnt = 0;
                  //printf("NSEC_BLOCK\n");
            }
            else
            {
                  checkEntriesInTcpUdpTable(UPDATE_CNT);
                  timer_cnt++;
                  //printf("NSEC\n");
            }
        }
        else if (signo == SIGINT) {
                timer_delete(timerid);
                perror("Ctrl + C cached!\n");
                exit(1);
        }
}

void getLocalMACAddress(char* dev)
{
    int s,i;
    struct ifreq ifr;

    s = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    strcpy(ifr.ifr_name, dev);

    ioctl(s, SIOCGIFHWADDR, &ifr);

    for (i=0; i<ETHER_ADDR_LEN; i++)
    {
        sprintf(&if_mac_str[i*2],"%02X",((unsigned
char*)ifr.ifr_hwaddr.sa_data)[i]);
    }

    if_mac_str[12]='\0';
}

void getInterfaceIPAddress(char* dev)
{
      int s;
      struct ifreq ifr;
      struct sockaddr_in *sin = (struct sockaddr_in *)
&ifr.ifr_addr;
      s = socket(AF_INET, SOCK_DGRAM, 0);

      strcpy(ifr.ifr_name, dev);

      sin->sin_family = AF_INET;

      ioctl(s, SIOCGIFADDR, &ifr);

      if_address=sin->sin_addr;
}

void sendTheRequestToPeer(struct in_addr dst_router, struct
in_addr ip_to_block)
{
      unsigned char destinationMAC[]="123167"; /* random mac
address */
      unsigned char len;

      /* some random ports */
      int sourcePort=1111;
      int destinationPort=2222;

      unsigned char* ip_to_block_str;
      unsigned char* finalPacket;
      unsigned char* userData;
      unsigned short totalLen;
      unsigned short udpTotalLen;
      unsigned short tmpType;
      unsigned int userDataLength;

      unsigned char command[]=ROUTER_COMMAND;
      unsigned char username[]=ROUTER_USERNAME;
      unsigned char password[]=ROUTER_PASSWORD;

      ip_to_block_str=strdup(inet_ntoa(ip_to_block));
      len=strlen(ip_to_block_str);

      encrypt(command, ENCR_KEY);
      encrypt(username, ENCR_KEY);
      encrypt(password, ENCR_KEY);
      encrypt(ip_to_block_str, ENCR_KEY);
```

```
	userDataLength=strlen(command) + strlen(username) + 
strlen(password)+len+1;
	userData = (unsigned char*)malloc( (userDataLength) * 
sizeof(unsigned char));

	memcpy((void*)userData,(void*)username,strlen(username));
	memcpy((void*)(userData + 
strlen(username)),(void*)password,strlen(password));
	memcpy((void*)(userData + strlen(username) + 
strlen(password)),(void*)command,strlen(command));
	memcpy((void*)(userData + strlen(username) + 
strlen(password)+strlen(command)),(void*)ip_to_block_str,len);

	finalPacket = (unsigned char*)malloc( (userDataLength + 
42) * sizeof(unsigned char));

	totalLen = userDataLength + 20 + 8;

	memcpy((void*)finalPacket,(void*)destinationMAC,6);
	memcpy((void*)(finalPacket+6),(void*)if_mac_str,6);
	tmpType = 8;
	memcpy((void*)(finalPacket+12),(void*)&tmpType,2);

	memcpy((void*)(finalPacket+14),(void*)"\x45",1);
	memcpy((void*)(finalPacket+15),(void*)"\x00",1);
	tmpType = htons(totalLen);
	memcpy((void*)(finalPacket+16),(void*)&tmpType,2);
	tmpType = htons(0x1337);
	memcpy((void*)(finalPacket+18),(void*)&tmpType,2);
	memcpy((void*)(finalPacket+20),(void*)"\x00",1);
	memcpy((void*)(finalPacket+21),(void*)"\x00",1);
	memcpy((void*)(finalPacket+22),(void*)"\x80",1);
	memcpy((void*)(finalPacket+23),(void*)"\x11",1);
	memcpy((void*)(finalPacket+24),(void*)"\x00\x00",2);
	memcpy((void*)(finalPacket+26),(void*)&if_address,4);
	memcpy((void*)(finalPacket+30),(void*)&dst_router,4);

	tmpType = htons(sourcePort);
	memcpy((void*)(finalPacket+34),(void*)&tmpType,2);
	tmpType = htons(destinationPort);
	memcpy((void*)(finalPacket+36),(void*)&tmpType,2);
	udpTotalLen = htons(userDataLength + 8);
	memcpy((void*)(finalPacket+38),(void*)&udpTotalLen,2);
	memcpy((void*)(finalPacket+40),(void*)&tmpType,2);
	memcpy((void*)(finalPacket+42),(void*)userData,userDataLe
ngth);

	pcap_sendpacket(handle,finalPacket,userDataLength + 42);
}


void sendTheBorderRouterMessage(struct in_addr dst_router, 
char* msg)
{
	unsigned char destinationMAC[]="123167";
	int sourcePort=1111;
	int destinationPort=2222;
	unsigned char* finalPacket;
	unsigned char* userData;
	unsigned short totalLen;
```

```
      unsigned short udpTotalLen;
      unsigned short tmpType;
      unsigned int userDataLength;

      userDataLength=strlen(msg);
      userData = (unsigned char*)malloc( (userDataLength) *
sizeof(unsigned char));


      memcpy((void*)userData,(void*)msg,strlen(msg));

      finalPacket = (unsigned char*)malloc( (userDataLength +
42) * sizeof(unsigned char));

      totalLen = userDataLength + 20 + 8;

      memcpy((void*)finalPacket,(void*)destinationMAC,6);
      memcpy((void*)(finalPacket+6),(void*)if_mac_str,6);
      tmpType = 8;

      memcpy((void*)(finalPacket+12),(void*)&tmpType,2);

      memcpy((void*)(finalPacket+14),(void*)"\x45",1);
      memcpy((void*)(finalPacket+15),(void*)"\x00",1);
      tmpType = htons(totalLen);
      memcpy((void*)(finalPacket+16),(void*)&tmpType,2);
      tmpType = htons(0x1337);
      memcpy((void*)(finalPacket+18),(void*)&tmpType,2);
      memcpy((void*)(finalPacket+20),(void*)"\x00",1);
      memcpy((void*)(finalPacket+21),(void*)"\x00",1);
      memcpy((void*)(finalPacket+22),(void*)"\x80",1);
      memcpy((void*)(finalPacket+23),(void*)"\x11",1);
      memcpy((void*)(finalPacket+24),(void*)"\x00\x00",2);
      memcpy((void*)(finalPacket+26),(void*)&if_address,4);
      memcpy((void*)(finalPacket+30),(void*)&dst_router,4);
      tmpType = htons(sourcePort);
      memcpy((void*)(finalPacket+34),(void*)&tmpType,2);
      tmpType = htons(destinationPort);
      memcpy((void*)(finalPacket+36),(void*)&tmpType,2);
      udpTotalLen = htons(userDataLength + 8);
      memcpy((void*)(finalPacket+38),(void*)&udpTotalLen,2);
      memcpy((void*)(finalPacket+40),(void*)&tmpType,2);
      memcpy((void*)(finalPacket+42),(void*)userData,userDataLe
ngth);

      pcap_sendpacket(handle,finalPacket,userDataLength + 42);
}


void encrypt(char* str,int key)
{
    unsigned int i;
    for(i=0;i<strlen(str);++i)
    {
          str[i] = str[i] - key;
    }
}

void decrypt(char* str,int key)
{
    unsigned int i;
```

```
    for(i=0;i<strlen(str);++i)
    {
          str[i] = str[i] + key;
    }
}

int isCommandFromFriends(char* ip_src_str)
{
      if( ((strcmp(ip_src_str, IP_ROUTER_A)==0) ||
(strcmp(ip_src_str, IP_ROUTER_B)==0) || (strcmp(ip_src_str,
IP_ROUTER_C)==0)) && (strcmp(ip_src_str, if_address_str) !=0 )
)
      {
            return 1;
      }

      return 0;
}

void executeRemoteCommand(const u_char* packet, int
payload_begin, char* ip_src_str)
{
      char iptables_block_ip_cmd_in[128];
      char iptables_block_ip_cmd_out[128];
      char border_router_message[128];

      unsigned char len;
      int i;
      struct in_addr tmp_addr;
      u_char* payload=(u_char*)(packet+payload_begin);
      char* username = (char*)malloc(strlen(ROUTER_USERNAME) *
sizeof(char));
      char* password = (char*)malloc(strlen(ROUTER_PASSWORD) *
sizeof(char));
      char* command = (char*)malloc(strlen(ROUTER_COMMAND) *
sizeof(char));
      char* ip_to_block;

      memcpy(username,payload,strlen(ROUTER_USERNAME));
      username[strlen(ROUTER_USERNAME)]='\0';

      memcpy(password,(payload+strlen(ROUTER_USERNAME)),strlen(
ROUTER_PASSWORD));
      password[strlen(ROUTER_PASSWORD)]='\0';

      memcpy(command,(payload+strlen(ROUTER_USERNAME)+strlen(RO
UTER_PASSWORD)),strlen(ROUTER_COMMAND));
      command[strlen(ROUTER_COMMAND)]='\0';

      memcpy((void*)&len,(payload+strlen(ROUTER_USERNAME)+strle
n(ROUTER_PASSWORD)+strlen(ROUTER_COMMAND)),1);

      ip_to_block=(char*)malloc(len*sizeof(char));

      memcpy((void*)ip_to_block,(payload+strlen(ROUTER_USERNAME
)+strlen(ROUTER_PASSWORD)+strlen(ROUTER_COMMAND)),len);

      decrypt(username,ENCR_KEY);
      decrypt(password,ENCR_KEY);
      decrypt(command,ENCR_KEY);
      decrypt(ip_to_block,ENCR_KEY);
```

```
	if ( inet_aton(ip_to_block, &tmp_addr) &&
(strcmp(username,ROUTER_USERNAME)==0) &&
(strcmp(password,ROUTER_PASSWORD)==0) &&
(strcmp(command,ROUTER_COMMAND)==0) )
	{

#ifndef IS_BORDER_ROUTER

		snprintf(iptables_block_ip_cmd_in, 127, "iptables -
I INPUT -s %s -j DROP", ip_to_block);
		snprintf(iptables_block_ip_cmd_out, 127, "iptables
-I OUTPUT -s %s -j DROP", ip_to_block);

	if( system(iptables_block_ip_cmd_in) != 0 )
		{
			perror("cannot execute iptable block command
!");
		}

		if( system(iptables_block_ip_cmd_out) != 0 )
		{
			perror("cannot execute iptable block command
!");
		}

		thresholdValue = thresholdValue / 2;

#else
		snprintf(border_router_message, 127, "Internal
network of %s has encountered worm
activity\n",external_interface);
		printf("%s\n",border_router_message);

		/* send the messages to the firends */
		for(i=0; i<number_of_friends; i++)
		{
			inet_aton(list_of_friends[i], &tmp_addr);
			sendTheBorderRouterMessage(tmp_addr,
border_router_message);
		}
#endif
	}
}

int isInTheWhiteList(char* ip_src_str, char* ip_dst_str, int
sport, int dport)
{
	int i = 0;
	unsigned char ok = 0;

	for(i=0; i<NUMBER_OF_WHITE_LIST_ELEMS; i++)
	{
		ok = 0;

		if((strcmp(whiteList[i].white_ip_src, "") == 0) ||
(strcmp(whiteList[i].white_ip_src, ip_src_str) == 0))
		{
			ok++;
		}
```

```
            if((strcmp(whiteList[i].white_ip_dst, "") == 0) ||
(strcmp(whiteList[i].white_ip_dst, ip_dst_str) == 0))
            {
                  ok++;
            }

            if((whiteList[i].white_sport == -1) ||
(whiteList[i].white_sport == sport))
            {
                  ok++;
            }

            if((whiteList[i].white_dport == -1) ||
(whiteList[i].white_dport == dport))
            {
                  ok++;
            }

            if( ok == 4 )
            {
                  return 1;
            }
      }

      return -1;
}
```

Makefile

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/


CC= gcc
CFLAGS= -O2
INCLUDES= -I. -I/usr/include/
LIBS= -lpcap -lresolv -lpjlib-util-i686-pc-linux-gnu -lpjnath-
i686-pc-linux-gnu -lpjsip-i686-pc-linux-gnu -lpjsip-simple-
i686-pc-linux-gnu -lpjsip-ua-i686-pc-linux-gnu -lpj-i686-pc-
linux-gnu -lrt
README=
EXEC= dns_sniff.o

all:    dns_sniff_func.c db.c Makefile
      $(CC) $(CFLAGS) $(INCLUDES) *.c -o $(EXEC) $(LIBS)

beauty:
      @indent -kr -i8 -ts8 -sob -l80 -ss -ncs *.[c,h];
      @rm -f *.[c,h]~;

clean:
      @rm -rf *.o *~ $(EXEC) core.* core

sense:
      @more $(README)
```

9.2 Pseudo-Worm Daemon (PWD) Source Code

UDPServers.c

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
#include <linux/if.h>
#include <netdb.h>

#define PHI 0x9e3779b9
#define OCTET_1_MIN 10
#define OCTET_1_MAX 11
#define OCTET_2_MIN 0
#define OCTET_2_MAX 0
#define OCTET_3_MIN 0
#define OCTET_3_MAX 0
#define OCTET_4_MIN 0
#define OCTET_4_MAX 0
#define SEED 1
#define NSECOND (100u)
#define MSECONDS (10u)
#define SECONDS (1000u)
#define N_MSECONDS (NSECOND * MSECONDS)
#define N_M_SECONDS (NSECOND * MSECONDS * SECONDS)
#define HardCodedStringLength 10
#define MAX_IP_LEN  15
#define IP_FROMFILE_MAXLEN 257
#define MAX_IP_ADDRESSES 10
#define MAX_RANDOM_IPS 357
#define GET_IPADDRESS_FROM_BYTE(a, b, c, d)\
    (((unsigned char)(a) << 24) & ((unsigned char)(b) << 16)\
&((unsigned char)(a) << 8) & ((unsigned char)(a)))

#define IP_ADDR_FILE
#define SEND_TO_RANDOM_IP

#define LOGGING_SERVER_PORT_NO (1600u)
#define LOGGING_SERVER_ADDRESS "127.0.0.1"

void getInterfaceInfo(char* dev);
void sendMsg(char* msg, struct in_addr ipAddr);
void sendLoggData(char* msg);

struct in_addr if_address;
struct in_addr if_bcastaddr;
struct in_addr if_netmask;

char buf[1024] = "";
char HardCodedStr[HardCodedStringLength + 1] = "";
```

```
int GetHardCodedString(char *str)
{
    FILE *fp;
    fp = fopen("HardCodedString", "r");
    if (NULL == fp)
    {
        printf("Open the HardCoded file failed,Exit...\n");
        return -1;
    }
    printf("Open the HardCoded file success\n");

    if (HardCodedStringLength != fread(str, sizeof(char),
HardCodedStringLength, fp))
    {
        printf("Read file failed,Exit...\n");
        fclose(fp);
        return -1;
    }

    fclose(fp);
    return 0;
}


int WriteTimelog()
{
      char buffer[30];
      char sendBuffer[60];
      struct timeval tv;
      time_t curtime;
      gettimeofday(&tv, NULL);
      curtime=tv.tv_sec;
      strftime(buffer,30,"%T:",localtime(&curtime));
      sprintf(sendBuffer,"%s%ld\n",buffer,tv.tv_usec);
      sendBuffer[strlen(sendBuffer)-2] = '\0';
      strcat(sendBuffer, " ");
      strcat(sendBuffer, inet_ntoa(if_address));
      sendLoggData(sendBuffer);
      return 0;
}

void CleanBuf(void)
{
    memset(buf, 0, sizeof(buf));
    memset(HardCodedStr, 0, sizeof(HardCodedStr));
}

static uint32_t Q[4096], c = 362436;

void init_rand(uint32_t x)
{
        int i;

        Q[0] = x;
        Q[1] = x + PHI;
        Q[2] = x + PHI + PHI;

        for (i = 3; i < 4096; i++)
                Q[i] = Q[i - 3] ^ Q[i - 2] ^ PHI ^ i;
}
```

```
uint32_t rand_cmwc(void)
{
        uint64_t t, a = 18782LL;
        static uint32_t i = 4095;
        uint32_t x, r = 0xfffffffe;
        i = (i + 1) & 4095;
        t = a * Q[i] + c;
        c = (t >> 32);
        x = t + c;
        if (x < c) {
                x++;
                c++;
        }
        return (Q[i] = r - x);
}

int main(int argc, char* argv[])
{
      int sock, i, numbytes;
      int tmp_A, tmp_B, tmp_C, tmp_D;
      int n_ipaddr;
      FILE *pIPAddr;
      struct sockaddr_in addrListen;
      struct sockaddr_in addrClient;
      struct sockaddr_in addrIPFile[MAX_IP_ADDRESSES];
      struct sockaddr_in addrIPLocalBroadcast;
      struct sockaddr_in addrIPRandom;
      int addrLength = sizeof(struct sockaddr_in);
      char IpFromFile[IP_FROMFILE_MAXLEN];
      char randomIp[IP_FROMFILE_MAXLEN];
      long number = 0;
      int RetIpFromFile = 0;
      int SendRet;
      struct timeval t1, t2;

      if(argv[1] == NULL)
      {
            argv[1] = strdup("eth0");
      }

      getInterfaceInfo(argv[1]);

    //while(1)
    {

      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if(-1 == sock)
      {
            printf("Create socket failed,Exit!\n");
            return -1;
      }

      printf("Create socket success and continue\n");

      memset(&addrListen, 0, sizeof(addrListen));
      memset(&addrListen, 0, sizeof(addrClient));
      addrListen.sin_family = AF_INET;
      addrListen.sin_addr.s_addr = INADDR_ANY;//local IP
Address
      addrListen.sin_port = htons(1434);
```

```
      if(-1 == bind(sock, (struct sockaddr*)&addrListen,
sizeof(addrListen)))
      {
            printf("Bind socket error,Exit...\n");
            return -1;
      }
      printf("Start listening Port 1434\n");

      memset(HardCodedStr, 0, sizeof(HardCodedStr));

      pIPAddr = NULL;

        recvfrom(sock, buf, 1024, 0, (struct
sockaddr*)&addrClient, &addrLength);
        number ++;
        printf("%ld : %s\n", number, buf);//display the content
of received packet

         if(GetHardCodedString(HardCodedStr) < 0)
        {
            printf("Get Authenticate string from file
failed\n");
                  CleanBuf();
                  exit(1);
        }
        printf("Get HardCoded String from file success\n");

    if ((strlen(buf) != HardCodedStringLength) ||(0 !=
strcmp(HardCodedStr, buf)))
      {
            printf("Authenticate failed\n");
            CleanBuf();
            //continue;
            exit(1);
      }
      else
      {
            printf("Authenticate success\n");
      }

         if (0 != WriteTimelog())
            {
                  printf("Write time log failed\n");
                  CleanBuf();
                  //continue;
                  exit(1);
            }
            else
            {
                  printf("Write time log success\n");
            }

      n_ipaddr = 0;

    if(pIPAddr == NULL)
      {
          pIPAddr = fopen("IPAddr", "r");
      }

        if(NULL == pIPAddr)
        {
```

```
            printf("Open IPAddr file failed\n");
          RetIpFromFile = -1;
        }
        else
        {
            while(1)
            {
                  memset(IpFromFile, 0, IP_FROMFILE_MAXLEN);

                  if (!(fgets(IpFromFile, MAX_IP_LEN + 1,
pIPAddr)))
                  {
                        //printf("Read IPAddr file failed\n");
                        //RetIpFromFile = -1;
                        break;
                  }
                  else
                  {
                        addrIPFile[n_ipaddr].sin_addr.s_addr =
inet_addr(IpFromFile);
                        RetIpFromFile = 0;
                        printf("Get IP Address from file
success: %s\n", inet_ntoa(addrIPFile[n_ipaddr].sin_addr));
                        n_ipaddr++;
                  }
            }
        }

      #ifdef IP_ADDR_FILE
      for(i=0; i<n_ipaddr; i++)
      {
            addrIPFile[i].sin_family = AF_INET;
            addrIPFile[i].sin_port = htons(1434);

            usleep(N_MSECONDS);

            SendRet = sendto(sock, HardCodedStr,
HardCodedStringLength, 0, (struct sockaddr*)&addrIPFile[i],
addrLength);

            if(-1 == SendRet)
            {
                printf("Send packet to random address (%s)
failed\n", inet_ntoa(addrIPFile[i].sin_addr));
            }
            else
            {
                  printf("Send packet to random address (%s)
success\n", inet_ntoa(addrIPFile[i].sin_addr));
            }
      }
      #endif

      #ifdef SEND_TO_RANDOM_IP
      if(sock)
      {
            close(sock);
      }

      init_rand(SEED ? SEED : time(NULL));
        while(1){
```

```
            gettimeofday(&t1, NULL);
      for(i=0; i<MAX_RANDOM_IPS; i++)
      {
        tmp_A = OCTET_1_MIN && (OCTET_1_MIN <= OCTET_1_MAX) ?
OCTET_1_MIN +
            (int) (rand_cmwc() % (OCTET_1_MAX - OCTET_1_MIN +
1)) : (int) (rand_cmwc() % 254);
        tmp_B = OCTET_2_MIN && (OCTET_2_MIN <= OCTET_2_MAX) ?
OCTET_2_MIN +
            (int) (rand_cmwc() % (OCTET_2_MAX - OCTET_2_MIN +
1)) : (int) (rand_cmwc() % 254);
        tmp_C = OCTET_3_MIN && (OCTET_3_MIN <= OCTET_3_MAX) ?
OCTET_3_MIN +
            (int) (rand_cmwc() % (OCTET_3_MAX - OCTET_3_MIN +
1)) : (int) (rand_cmwc() % 254);
        tmp_D = OCTET_4_MIN && (OCTET_4_MIN <= OCTET_4_MAX) ?
OCTET_4_MIN +
            (int) (rand_cmwc() % (OCTET_4_MAX - OCTET_4_MIN +
1)) : (int) (rand_cmwc() % 254);


            sprintf(randomIp, "%d.%d.%d.%d", tmp_A, tmp_B,
tmp_C, tmp_D);

            sprintf(randomIp, "%d.%d.%d.%d", tmp_A, tmp_B,
tmp_C, tmp_D);

            inet_aton(randomIp, &addrIPRandom.sin_addr);

            sendMsg(HardCodedStr, addrIPRandom.sin_addr);
            usleep(N_M_SECONDS / MAX_RANDOM_IPS);
      }
#define USEC 100000
            gettimeofday(&t2, NULL);
            int sec = t2.tv_sec - t1.tv_sec;
            int msec = (t2.tv_usec - t1.tv_usec) / USEC;
            if (t2.tv_usec < t1.tv_usec)
            {
                sec--;
                msec = (t2.tv_usec + 1000000 - t1.tv_usec) /
USEC;
            }
            printf("%d packets were send during %u.%u
seconds\n", MAX_RANDOM_IPS, sec, msec);
      }
      #endif
  }

    fclose(pIPAddr);

    return 0;
}

void getInterfaceInfo(char* dev)
{
      int s;
      struct ifreq ifr;
      struct sockaddr_in *sin = (struct sockaddr_in *)
&ifr.ifr_addr;
      s = socket(AF_INET, SOCK_DGRAM, 0);
      strcpy(ifr.ifr_name, dev);
```

```
      sin->sin_family = AF_INET;
      ioctl(s, SIOCGIFADDR, &ifr);
      if_address=sin->sin_addr;
      ioctl(s, SIOCGIFNETMASK, &ifr);
      if_netmask = sin->sin_addr;
      ioctl(s, SIOCGIFBRDADDR, &ifr);
      if_bcastaddr = sin->sin_addr;
}


void sendMsg(char* msg, struct in_addr ipAddr)
{
    int sockfd;
    struct sockaddr_in their_addr;
    int numbytes;
    //char broadcast = '1';

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;
    their_addr.sin_port = htons(1434);
    their_addr.sin_addr = ipAddr;
    memset(their_addr.sin_zero, '\0', sizeof
their_addr.sin_zero);

    if ((numbytes=sendto(sockfd, msg, strlen(msg), 0, (struct
sockaddr *)&their_addr, sizeof their_addr)) == -1)
    {
        perror("sendto");
        exit(1);
    }

    //printf("Message %s sent to %s\n", msg,
inet_ntoa(their_addr.sin_addr));

    close(sockfd);
}

void sendLoggData(char* msg)
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    portno = LOGGING_SERVER_PORT_NO;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(LOGGING_SERVER_ADDRESS);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char
*)&serv_addr.sin_addr.s_addr, server->h_length);
    serv_addr.sin_port = htons(portno);
```

```
    if (connect(sockfd,(struct sockaddr *)
&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");

    n = write(sockfd,msg,strlen(msg));
    if (n < 0)
         error("ERROR writing to socket");

    close(sockfd);
}
```

UDPClient.c

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

char szMsg[] = "teststring";

int main(int argc, char* argv[])
{
    int sock;
    int uIndex = 1;
    struct sockaddr_in addrto;
    int nlen = sizeof(addrto);
    int AddrRet;

    printf("Start running\n");

    if(argc != 2)
    {
        printf("Number of parameter error! Exit...\n");
        return -1;
    }

    memset(&addrto, 0, sizeof(addrto));
    addrto.sin_family = AF_INET;
    //addrto.sin_addr.s_addr = inet_addr("127.0.0.1");
    AddrRet = inet_aton(argv[1], &addrto.sin_addr);
    if(0 == AddrRet)
    {
        printf("IP Address Parameter wrong! Exit...\n");
        return -1;
    }
     addrto.sin_port = htons(1434);
    printf("Set the destination address success and
continue\n");

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if(-1 == sock)
    {
        printf("Create socket failed,Exit!\n");
        return -1;
    }
    printf("Create socket success and continue\n");
```

```
    //while(1)
    //{
        sendto(sock, szMsg, strlen(szMsg), 0, (struct
sockaddr*)&addrto, nlen);
        printf("%d : an UDP package send\n", uIndex++);
        sleep(5);
    //}

    close(sock);

    return 0;
}
```

LoggingServer.c

```
/* (c) Copyright University of Greenwich 2015 /*
/*http://www.gre.ac.uk/isrl*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT_NO (1600u)

void dumpLogData(int);
void error(const char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, pid;
     socklen_t clilen;
     struct sockaddr_in serv_addr, cli_addr;

     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = PORT_NO;
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     while (1) {
         newsockfd = accept(sockfd,
               (struct sockaddr *) &cli_addr, &clilen);
         if (newsockfd < 0)
             error("ERROR on accept");
         pid = fork();
```

```
         if (pid < 0)
             error("ERROR on fork");
         if (pid == 0)  {
             close(sockfd);
             dumpLogData(newsockfd);
             exit(0);
         }
         else close(newsockfd);
     } /* end of while */
     close(sockfd);
     return 0;
}

void dumpLogData (int sock)
{
   int n;
   char buffer[256];
   FILE *fp;
   fp = fopen("Timelog", "a");
   if(NULL == fp)
   {
      printf("Open Timelog file failed, Exit...\n");
        exit(1);
   }

   bzero(buffer,256);
   n = read(sock,buffer,255);
   if (n < 0) error("ERROR reading from socket");
   printf("Log created: %s\n",buffer);

   fwrite(buffer, sizeof(char), n, fp);
   fputs("\n", fp);

   fclose(fp);
}
```

IPAddr.txt

```
10.63.2.11
10.18.56.78
10.128.3.4
10.2.21.43
10.4.6.7
```

Timelog.txt

```
17:26:39:95432 10.63.2.11
17:43:28:53889 10.18.56.78
17:46:02:95361 10.128.3.4
17:47:03:31260 10.2.21.43
17:48:05:65620 10.4.6.7
```